1 2 9 0

UNIVERSIDADE Ð COIMBRA

Gonçalo de São José Marques da Silva

Sistema Inovador para a comunicação entre uma mãe e o seu bebé prematuro, numa incubadora

Dissertação no âmbito do Mestrado Integrado em Engenharia Eletrotécnica e de Computadores orientada pelos Professores Doutores Jorge Miguel Sá Silva e André Miguel de Almaida Marrão Rodrigues e apresentada ao Departamento de Engenharia Eletrotécnica e de Computadores da Faculdade de Ciências e Tecnologia da Universidade de Coimbra.

Outubro de 2021

FCTUC

DEEC
DEPARTAMENTO DE ENGENHARIA ELECTROTÉCNICA E DE COMPUTADORES

Development of a Prototype of a System for the communication between a Mother and a Premature Baby in an Incubator

Author:
Gonçalo Silva
uc2014196747@student.uc.pt

Supervisors:
Prof. Dr. Jorge Sá Silva
Prof. Dr. André Rodrigues

Jury:
President: Prof. Dr. Paulo José Monteiro Peixoto
Supervisor: Prof. Dr. Jorge Sá Silva
Vogal: Prof. Dr. Alberto Jorge Lebre Cardoso

October, 2021

Acknowledgements

Gostaria de começar este documento por mencionar e agradecer a todas as pessoas que me acompanharam ao longo do meu percurso académico.

Em primeiro lugar agradecer aos meus orientadores, Professor Jorge Sá Silva e Professor André Rodrigues, pela oportunidade de realizar este projeto, assim como todo o apoio e orientação durante o seu desenvolvimento.

De seguida, gostaria de mencionar amigos e colegas que conheci e com quem partihei esta viagem. Diogo, João e Catarina, estiveram presentes nos bons e nos maus momentos e ajudaram a tornar estes anos especiais.

Guardado para o fim fica um enorme agradecimento e sentimento de apreciação para com a minha família. Ninguém escolhe onde nasce e eu não podia ter tido mais sorte. Ao longo destes anos os meus pais, irmãos e avós estiveram sempre presentes, com apoio incondicional e preocupação. Obrigado pelas oportunidades que me deram e por serem o meu suporte ao longo destes anos.

Resumo

De acordo com vários estudos, a separação entre mãe e bebé nos primeiros meses de vida pode ter um impacto severo na sua relação e trazer consequências negativas para ambos a nível psicológico. Este cenário é particularmente prevalente quando o bebé nasce prematuro e é colocado numa incubadora. Os últimos meses de gravidez são importantes para estabelecer uma conexão entre os dois, uma vez que o bebé pode ouvir os sons vitais da mãe e estão os dois em contacto permanente. Tendo isto em conta, este projeto pretende desenvolver, em parceria com a Faculdade de Psicologia e Ciências da Educação da Universidade de Coimbra, um protótipo de um sistema que procura mitigar estas consequências. Implementando técnicas de telecomunicação, o protótipo será capaz do envio dos sons uterinos da mãe, em tempo real, ao bebé. Ao mesmo tempo, será também capaz do envio de vídeo e áudio do bebé para a mãe. Para o efeito o protótipo será dividido em duas partes, uma implementada do lado do bebé e outra do lado da mãe.

Abstract

According to several studies, early separation of mother and child can have a severe impact on their relationship, leading to negative consequences on the psychological health of both. This scenario is very prevalent when babies are born prematurely and placed in an incubator. The final months of pregnancy are very important to establish a bond between the two, as the baby can hear the mother's vital sounds and they are both in permanent contact. With all this in mind, and in partnership with the Faculdade de Psicologia e Ciências da Educação da Universidade de Coimbra, this project looks to develop a prototype of a system that can mitigate these consequences. By implementing telecommunication techniques, the prototype shall be able to provide womb sounds, in real time, to the baby. At the same time, a video and audio stream is sent from the incubator to the mother. The prototype will, therefore, be divided in two components, one of them assembled by the incubator and the another, that accompany the mother.

Keywords— Internet of Things, Incubator, Mobile Development, WebRTC, Multimedia Streaming, Internet Protocols

Contents

List of Figures

Glossary

ACME	Automatic Certificate Management Environmen.
ALSA	Advanced Linux Sound Architecture.
API	Application Programming Interface.
ARM	Advanced RISC Machines.
BLE	Bluetooth Low Energy.
CA	Certificate Authority.
CSR	Certificate Signing Request.
DNS	Domain Name System.
HTTP	Hypertext Transfer Protocol.
HTTPS	Hypertext Transfer Protocol Security.
ICE	Interactive Connectivity Establishment.
IoT	Internet of Things.
JNI	Java Native Interface.
JSON	JavaScript Object Notation.
JVM	Java Virtual Machine.
NAT	Network Address Translation.
NDK	Native Development Kit.
NICU	Neonatal Intensive Care Unit.
P2P	Peer to Peer.
PIV	Personal Identity Verification.
RTP	Real-time Transport Protocol.
RTSP	Rapid Spanning Tree Protocol.
SDK	Software Development Kit.
SDP	Session Description Protocol.
SSL	Secure Sockets Layer.
STUN	Session Traversal Utilities for NAT.
TCP	Transmission Control Protocol.
TURN	Traversal Using Relays around NAT.
UDP	User Datagram Protocol.
UI	User Interface.
URL	Uniform Resource Locator.
VLC	VideoLAN Client.

WebRTC	Web Real-Time Communications.
WS	WebSocket.
WSS	WebSocket Secure.
XAML	Extensible Application Markup Language.

1 Introduction

1.1 Framing and motivation

Every year 1 in 10 babies worldwide is born prematurely. In Portugal that number is around 8% [1]. Besides the widely known and studied physical consequences of a premature birth there are also a lot of problems regarding the psyche for both the mother and baby.

Due to a fragile physical condition, the newborn and premature baby requires a special habitat, that can replicate, to an extent, the conditions of the womb, i.e. an incubator, which is usually located in a NICU (Neonatal Intensive Care Unit). Placing the baby in such a device will lead to the physical separation between mother and baby. Consequently the mother can become cold towards the baby and even develop symptoms of depression. As for the baby, the lack of contact, either physical or just hearing the mothers voice, gut sounds and heartbeat can lead to irregular and unpredictable behavioral patterns at early ages and developmental problems later on [2]. The sounds of the heartbeat, gut and voice are very important for the baby's connection with the mother. Hearing womb sounds is not only good for the connection between mother and baby, but also important in the baby's late stages of development inside the womb.

These sounds are also different from person to person and, throughout the day, unpredictable. The mother's change in state of mind has a big role in the mother and baby's connection. Therefore it is important to provide these sounds to the baby in real time so that he can relate to the mothers state of mind whenever it changes. Hearing these sounds alone can mitigate the negative consequences of separation and improve the bond between mother and baby.

1.2 Objectives

This project looks to demonstrate the implementation of a prototype of a system that can improve the relation between babies in incubators and their mothers, as well as contribute to their healthy development, hopefully leading to the mitigation of the consequences described.

As previously stated, this separation is characterized by the absence of the womb sounds the baby would hear in the final months of pregnancy and the lack of contact that both subjects endure after birth. Therefore, the prototype looks to provide the baby with the womb sounds he would hear in the final stages of pregnancy and the mother with a multimedia stream that allows her to watch the baby whenever she wishes.

The prototype looks to do this by implementing an IoT (Internet of Things) system that can support bidirectional real-time communication between the incubator and the mother. It intends to implement two devices, one that can be mounted next to an incubator and support the streaming of different types of media in both directions simultaneously; and another that accompanies the mother and supports the streaming of womb sounds.

It also looks to implement an app which facilitates access to the multimedia stream coming from the incubator.

This prototype is merely a proof of concept. In order to be implemented in an hospital with willing subjects, it legally must follow a set of standards and rules that dictate how medical devices are set up. These standards and rules exist to ensure safe and secure implementation of medical devices. Following the medical industries' standards is outside the scope of this project.

1.3 Document Structure

This document is divided in seven main chapters, each divided into more detailed subsections.

Initially this document presents some of the concepts that come into play later, as well as scientific work that explores similar ideas. Knowing that the objective of this work is to build an actual prototype, it was determined that it would be important to find scientific work and patents that may have found solutions to similar problems, or dove into some of the issues we intend to mitigate. This information can be found in chapter 2.

Chapter 3 describes the project's architecture, the prototype's requirements, both functional and non-functional and details the equipment behind it.

Chapter 4 dives into specific hardware and software used in this prototype, which requires some deeper understanding.

Chapter 5 provides an overall look at the methodology for building the prototype and the planning done from the beginning. The risks and mitigation strategies are also explored in this chapter.

Chapter 6 looks deeper into the work that was done throughout this project.

Chapter 7 will also evaluate whether the risks where well predicted and the respective mitigation strategies where properly thought out.

2 State of the Art

2.1 Concepts

Throughout the duration of this project we tread upon several concepts. The one that stands out the most is *Internet of Things (IoT)*, because of the use of sensors, cameras and microphones. These devices interact with the real world and carry it over to the digital world, one of the main ideas behind *IoT*, providing interaction at a distance by two or more subjects.

We focus on *Mobile Development*, as an app is developed to support the multimedia stream.

Several *Internet Protocols* are explored and implemented. Some of them are very prevalent in the field of Network Engineering and a common presence in Internet Communications. Others are specific to multimedia streaming across a network.

Finally, the need to secure the information being exchanged, knowing that it can be considered sensitive, requires exploration of the concepts of *Privacy* and *Security*, by the implementation of encryption algorithms and safe servers.

2.2 Related Scientific Work

The work done by Kimberly Howard, Anne Martin, and Jeanne Brooks-Gunn of the Columbia University and Lisa J. Berlin of the Duke University [2] shows us a study of the consequences of early mother-child separation, both on the child's well-being and the stability of the family. It states that children who endure early separation from the mother may suffer from insecure attachment and subsequent mental problems. This separation may even lead to violent behavior in later years as well as difficulty for the child to connect with others.

The article by Bremmer, P., Byers, J. and and Kiehl, E., published on the Journal of Obstetric, Gynecologic, and Neonatal Nursing [3] analysis research on the physical and psychological effects of noise on the premature baby's health. It lists the adverse effects of loud noise as being "apnea, bradycardia, as well as abrupt fluctuations in heart rate, respiratory rate, blood pressure and oxygen saturation". It also describes increased risk for hearing loss and attention-deficit hyperactivity disorder.

The work done by Joanna J. Parga, Robert Daland, Kalpashri Kesavan, Paul M. Macey, Lonnie Zeltzer and Ronald M. Harper looks to *"assess the efficacy and feasibility of external and non-invasive recordings in pregnant women"* [4]. This study attempts to record womb sounds at several stages of pregnancy and compares them to commercially available womb sounds. It concludes that external non-invasive recordings of womb sounds show a predominance of low frequencies and bowel sounds, distinct from popular commercial products available for new parents. This paper shows that collecting womb sounds directly from the mother is the better method for providing a closer experience in the incubator to an actual womb.

2.3 Patents

While doing research for this project we came across patents that tried to tackle the same issues. Below, patents of systems for communication in incubators as well as incubators with specific characteristics are listed.

Devroey, Edmond M., "Uterine Sound and Motion Simulation Device" International Patent 20,130,096,368, issued April 18, 2013 [5] is a patent for a device that simulates the movement as well as the sounds a baby would experience in the uterus.

For those purposes it uses bladders and pillows to simulate respiration and gait movements. Through the use of computer controlled operations it is able to simulate the sounds of the mothers voice and her body sounds as well as the their intensity throughout the day. It also claims to be of easy implementation in any incubator.

The main difference between this patent and our innovative system for communication between a mother and premature baby in an incubator is the use of pre-recorded sounds and their simulation based on those recordings.

Keisuke Wakabayashi (Saitama), Masato Honda (Saitama), Yutaka Sekiguchi (Saitama), Ichiro Matsubara (Tokyo), Terumi Matsubara (Tokyo), inventors; Atom Medical Corporation (Tokyo), assignee. "Incubator" US Patent 10,245,199, issued Apr 2, 2019 [6] is a patent which claims to provide an incubator that is more simple to use, having a latch that can move back and forth through an operable member. Safety precautions are provided so that the latch does not open in the off chance an unpredictable external force is applied to the operable member.

Taek Kyu Lee, La Mirada, CA(US); Min Soo Han, Irvine, CA(US), inventors. "Baby Monitoring System" US Patent US8,094,013B1, issued Jan. 10, 2012 [7] is a patent for a baby monitoring system that comprises a parent union and a sensor unit. The sensor unit contains two tri-axial accelerometers positioned so that they can independently monitor the baby's position and breath rate. The sensor unit sends the data to the parent unit which displays it on a screen. This system claims to be capable of recording the mother's heartbeat and sending it to the baby. However this process does not occur in real time.

Ronald S. Kolarovic, Cinnaminson,NJ (US); Barry E. Barsky, Huntington, PA (US), inventors; Hill-Rom Services, Inc., Batesville, IN (US), assignee. "Infant Incubator with Non-Contact Sensing and Monitoring" US Patent 6,679,830 B2, issued Feb. 6, 2002 [8] regards a patent for an infant care unit. It claims to comprise one or more sensors apart from the infant to sense a psychological parameter, from which it derives the best decision to control the incubator's environment. These may include respiration and temperature sensors. It also claims to include a microphone, camera and speakers to monitor the baby, provide noise cancellation or play sounds to the baby.

2.4 Analyzing Existing Solutions

Out of the works just mentioned, [5] attempts to implement a similar idea to the one present in this document by providing simulated sounds of the mother's body and voice. [4] seems to favor the idea of providing real sounds to the baby, instead of commercially available ones, which are disconnected from the mother and overly manipulated. However, they are not collected in real time. They are previously recorded and then played to the baby in a loop. This project intends to go further by providing, not only real sounds, but real sounds in real time, offering a clear advantage to [4] and [5].

3 Architecture

3.1 Description

This prototype consists of the modules described in figure 1. Module 1 relates to the device responsible for collecting womb sounds from the mother, in real time. These womb sounds are sent to module 2, which is responsible for redirecting them to the incubator. Module 3 receives these sounds and plays them to the baby. These three modules describe the direction in which the womb sounds travel: from the mother, after being collected, to the incubator, where they are provided to the baby.

The multimedia stream from the incubator and its path is represented by the remaining modules. Module 4 represents video and audio gathering. Module 5 is responsible for synchronizing video and audio into a stream and sending it to the mother. Module 6 provides the mother easy access to the stream.

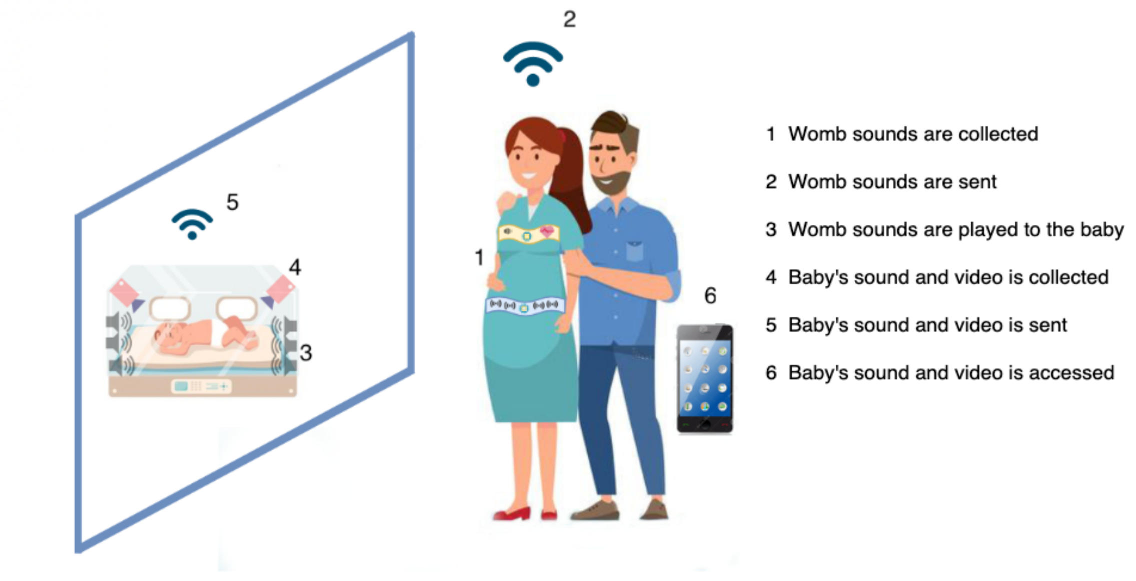
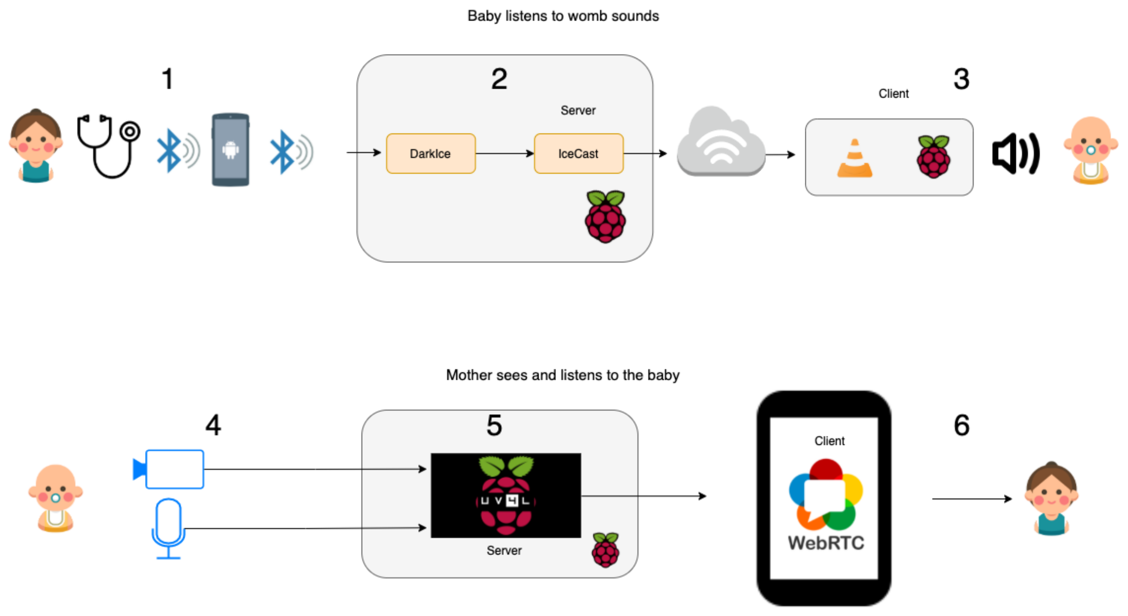

2
5
4
3
1
6
1 Womb sounds are collected
2 Womb sounds are sent
3 Womb sounds are played to the baby
4 Baby's sound and video is collected
5 Baby's sound and video is sent
6 Baby's sound and video is accessed

Figure 1: Model with modules used in the project.

Figure 2 represents the project's high level architecture. In the upper section, "Baby listens to womb sounds", the first 3 modules are included. Module 1 is represented by the stethoscope and smartphone. These two are responsible for collecting the womb sounds in real time and sending them to Module 2, which is a Raspberry Pi. Inside it, two modules, DarkIce, a piece of software responsible for audio encoding and streaming, and IceCast, an audio server, send the audio via the Internet to another Raspberry Pi running VLC (VideoLan Client). This second Raspberry Pi represents module 3 and is connected to speakers which provide the baby with the audio.



Baby listens to womb sounds
1
2
Server
DarkIce
IceCast
Client
3
Mother sees and listens to the baby
4
5
U V 4 L
Server
Client
WebRTC
6

Figure 2: Visual Representation of Project Architecture.

Modules 4, 5 and 6 are represented by the lower section of figure 2, "Mother sees and listens to the baby". Module 4 is represented by the devices responsible for collecting the baby's stream. Module 5 is a Raspberry Pi running UV4L, a generic purpose Streaming Server plug-in, especially made for IoT devices [9]. This Raspberry Pi is the same responsible for providing the baby with the womb sounds. Module 6 is a smartphone running an app that facilitates access to the stream. This app supports WebRTC (Web Real-Time Communications), a framework for real-time online communication.

Figure 2 does a good job of providing a solid understanding of how modules 1, 3, 4, 5 and 6 are connected, as the "Mother sees and listens to the baby" side of the architecture is simple and its complexity is due to implementation. Module 2, however, benefits from a deeper dive into its architecture.

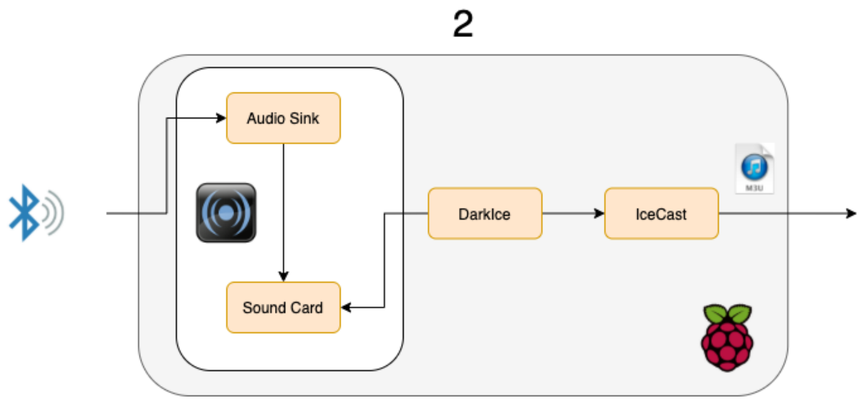


Figure 3: Visual Representation of the Architecture of Module 2.

Module 2's architecture is dependent on ALSA (Advanced Linux Sound Architecture), the framework which provides Linux audio functionalities. Inside the Raspberry Pi, PulseAudio, a piece of software capable of controlling ALSA for several audio applications, is responsible for managing the Raspberry Pi's audio bluetooth connection. It stores the audio in an Audio Sink, a destination for sound within ALSA and redirects it to a Sound Card. Sound Card is the name given to sources of audio in ALSA. DarkIce accesses the Sound Card and encodes the audio, redirecting it to IceCast, where it will be available as a m3u file. This format supports several types of media streaming in the Internet.

3.2 Functional Requirements

The prototype shall be able to provide the mother's womb sounds to the baby for as long as possible. This translates into implementing a solid software framework which can function reliably. The duration of the womb sounds stream must only be dictated by the digital stethoscope's battery life and the mother's willingness to use it.

The multimedia stream, coming from the incubator, shall be accessible whenever the mother wants to see her baby. Therefore, the communication between the Raspberry Pi by the incubator and the mother's smartphone, running the WebRTC app, must be established when required.

The hardware shall be as noninvasive as possible, so not to disturb the incubator's environment. This means choosing material that can be assembled outside the incubator and still provide the needed functionalities.

3.3 Non-Functional Requirements

The multimedia stream shall have enough quality so that the baby is clearly observable by the mother.

The audio the baby listens to shall be as close to what he would hear inside the womb as possible.

The prototype shall provide a minimum level of privacy, so that the womb sounds and baby's stream are only accessible by the subjects.

4 Technology

4.1 Equipment

Cameras In order to record the video and send it in real time to the mother a camera was required.

When connecting to the Raspberry Pi several connection and compatibility issues must be taken into account. With that in mind, the camera is the RaspiCam. This camera is developed to be compatible with a Raspberry Pi.

Microphones As stated before, when working with incubators it is important not to disturb its environment. Therefore the microphone must be able to record through the wall of the incubator. For that purpose a contact microphone that can be attached to the side of the incubator is being used.

It works by detecting the vibrations of sounds hitting whatever surface it is in contact with and amplifying them. If those vibrations are too low the gain can be increased via a dial located on the back of the device.

Speakers For the baby to hear the sound from the mother it was taken into account that the incubator is a hermetically closed environment. Therefore we decided to go for a device designed to send sound through surfaces. The chosen speakers are designed for unborn babies to listen to music.

Electronic Stethoscopes To record the mothers sounds the solution is to use a body strap with some sort of recording device attached to it. The use of small microphones was considered but the likely presence of noise from the grinding of clothes and the strap itself with the body led us to consider a digital stethoscope. Several of these are available in the market with straps included.

Most of these devices are very expensive. That made it easier to choose the Stemoscope™, the most affordable option with all the desired functionalities. it is small, light and allows for 5 hours of continuous use. It can be connected to any device that supports BLE (Bluetooth Low Energy) technology. An app accompanies this product, and the Stemoscope's use is limited to it.

Raspberry Pi The Raspberry Pi is a small single-board computer that can be used in IoT systems. This project required two and the particular model used for the prototype is a Raspberry Pi3 Model B. They are responsible for long distance communication. The one on the mother's side must send the audio collected with the digital stethoscope. It is connected to a smartphone via Bluetooth from which it receives the audio and then sends it.

The one on the baby's side must support several devices that are connected to it, collecting the video and audio using the RaspiCam and contact microphone and sending that to the mother. It also receives the mother's sound from the other Raspberry Pi and transmits it to the baby through the speakers.

4.2 Software and Frameworks

4.2.1 PulseAudio

PulseAudio is a sound server system for sound applications within the Raspberry Pi. "*It is an integral part of all relevant modern Linux distributions and is used in various mobile devices, by multiple vendors* [10]."

PulseAudio has a diverse set of loadable modules, each with its own set of functions, making it an extremely versatile tool for audio management, particularly in Linux devices. It does this by taking advantage of ALSA's API (Aplication Programming Interface) for Sound Cards device drivers.

4.2.2 DarkIce

DarkIce is a live audio streamer. It records audio from an audio interface (e.g. sound card), encodes it and sends it to a streaming server. It supports several audio interfaces (ALSA, Solaris, Jack, and others). Raspberry Pi's operating system is the Raspbian, which is Debian GNU/Linux based operating system. Like most Linux distributions, Raspbian uses ALSA, a software framework that provides an API for all sound management in the operating system. DarkIce can also send audio to several servers, including IceCast, which will be described below.

4.2.3 IceCast

IceCast is an open source server that supports audio streaming. It works over standard HTTP (HyperText Transfer Protocol) and supports MP3 streaming. It provides a m3u file through its UI (User Interface) that can be played by a regular media player like VLC.

4.2.4 VLC

VLC is a multipurpose media player that allows its users to play most types of media. It is free, open-source and cross-platform, making it available on almost any device.

VLC also allows a user to stream video and audio in real time. It is supported by most platforms, including mobile devices and Raspbian. Using the option "Open Capture Device", the user can then choose the input method and the type of codec for the stream. It supports HTTP, RTP (Real-time Transport Protocol), RTSP (Rapid Spanning Tree Protocol) and others. A link is then provided which points to the stream and can be shared. In another device, with the option "Open Network Stream" a user can use the link to start watching the stream.

4.2.5 App and Audiovisual stream

This prototype requires the development of an app that can function, preferably, on several platforms and allow for the implementation of the prototype with any candidate, regardless of the device type they might possess. With that in mind a platform and language that allows for Android and IOS development is used.

The most interesting options for multiplatform development are Xamarin, React Native, Flutter and Kotlin. According to [11] React Native operates on JavaScript, Flutter on Dart and Kotlin Multiplatform on Kotlin. Xamarin, being a part of the .NET platform, operates on C#. Furthermore, all these languages require knowledge of Swift for IOS development, regarding certain platform specific behaviors. Among all these, the only language that provides Native development and full Java interoperability is Kotlin Multiplatform, making it the most interesting option for faster and more reliable development. Xamarin, on the other end, allows simple IOS and Android apps to share up to 95% of their code.

Therefore, Kotlin Multiplatform seems to be the most promising app-development framework. It requires the least learning of all the options and, if any setbacks with IOS development or lack of Kotlin Multiplatform resources online arise, provides the possibility of focusing merely on Android development without having wasted any time. This is due to its interoperability with Java and ability to import Native Android libraries directly into a project.

As for the audiovisual stream there are few options that support this type of IoT applications for the Raspberry Pi. WebRTC is the most common for this purpose and offers the most online support.

4.2.6 Xamarin and Library Binding

Initially the App was meant to be developed in Xamarin. Xamarin, as a part of the .NET platform, deals mainly with C# (UIs are developed using XAML - Extensible Application Markup Language). It manages to use a single language for app development, while maintaining most of it Native, resulting in better performance and easier maintenance. Xamarin achieves this by providing several libraries specifically for this purpose, bridging the gap between C# and Native development very efficiently.

Using those libraries, Xamarin presents serious advantages. On one end, the UI code can be shared by several platforms, from Android, to IOS and Windows, using Forms technology. This allows for 90 up to 95% of the front and back-end code of an app to be shared cross-platform [11]. This also leads to faster and cheaper development, ideal for apps that require little platform specific functionality.

Opposed to Forms, Xamarin Native provides an opportunity for app development that can share back-end code, but requires native behavior or a platform custom UI. Although this leads to slower and more expensive development, performance is generally better and allows for more complex apps . Even though more components are specific to each platform, they can still share up to 70% of the code [11].

Due to not being familiar with C#, object oriented languages and app development, a significant portion of the initial stages of developing this app were focused on learning the language, as well as trying to understand how to implement something like a WebRTC library in a Xamarin project. As a starting point, a WebRTC app, developed in Xamarin and available on GitHub, providing a simple video and audio loopback using WebRTC methods, was used as the initial framework.

WebRTC is not among one of the many Native libraries that Xamarin offers. WebRTC mobile libraries are developed using Java, and, therefore, not compatible with the .NET platform. However, through a process called library binding, Xamarin allows for the creation of a Bindings library that automatically wraps the needed Java library with C# wrappers; pieces of code that can adapt code from a different language into, in this context, C#; so you can invoke Java code in C#. A WebRTC open-source library binding was already available online.

Library binding is a really useful tool but it does have its problems. To begin with, bindings aren't provided for every Java Library, and the ones that exist might not wrap every Java method and member. Not only that, but bindings aren't upgraded automatically. This means that deprecated method wrappers will not be removed and new methods will not be wrapped. As a result, bindings can easily become outdated.

New Java updates can also compromise library bindings. For safeguard the .NET platform implements functionalities like *"desaguring"*, which disables Java's *"syntactic sugar"* tool. This allows older bindings to stay functional. However, this tool can also affect newer bindings.

Errors like *"Can't call Java static interface methods in android libraries"* and *Java.Lang. NoSuchMethodError: 'no static method "Lorg/webrtc/EglBase;.create()Lorg/webrtc/EglB ase;"'* prevented the app from functioning properly. The app still ran but the stream was not available. Online solutions pointed towards deactivating the Java *"desaguring"* tool. However, this is not only highly inadvisable, as it can lead to other problems with the bindings, but also .NET and Visual Studio do not provide a clear and obvious way to deactivate it. Such changes might have required changing Java definitions, creating a certain amount of risk to the computers performance.

The previously stated error, even though it did not point to the specific problem with the binding, it did state which methods were not available for usage: the *"EglBase.create"* method. EGL is a cross-platform API that handles graphics management. WebRTC uses EGL as tool for video display in its *"SurfaceViewRenderer"* class. Xamarin provides an interface called JNI (Java Native Interface), which allows a developer to wrap specific Java classes and use with C#. However, after some research, this method revealed itself to be quite cumbersome and risky, not providing guarantees the wrappers would function properly, or if any other class or method from the WebRTC Native Library would require the same treatment.

It should be noted that it is possible to make your own library bindings, but it is a challenging process for someone with little experience in .NET and C#. It also requires the installation of many developer tools only available in Linux. These tools also require some understanding. It was decided that the effort was not worth the risk of not being able to do it or the result not answering our problems, since there was no guarantee our bindings would work.

4.2.7 Kotlin

Kotlin is a cross-platform programming language designed to interoperate fully with Java. This is done by making the JVM (Java Virtual Machine) of Kotlin's standard library depend on the Java Class Library. However Kotlin presents a much more concise and friendly syntax when compared to Java. In 2019 Kotlin was announced to be Google's primary choice for Android development. Another of Kotlin's advantages are Coroutines, a powerful tool for asynchronous and non-blocking programming, faster than traditional threads.

Kotlin also provides a Kotlin Mobile SDK (Software Development Kit), which allows for IOS and Android apps to share the business logic. This is called Kotlin Multiplatform. The secret for Kotlin's reliability is its flexibility, meaning that developers have lots of freedom in what to share between IOS and Android. Figure 4 shows how common code can be shared and interact with the Kotlin JVM, JS (JavaScript) and Native libraries for several platforms.

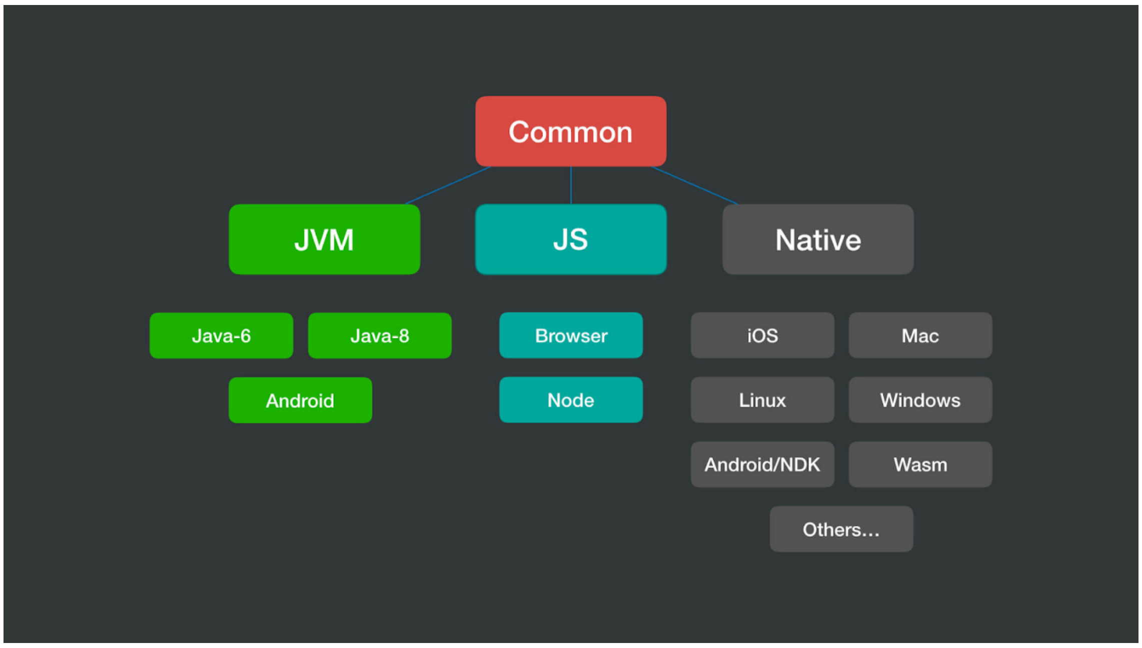


Figure 4: Interaction between the different Kotlin modules [11]

A Kotlin Multiplatform project has several modules, one containing the Android code, another containing the IOS code and a third one containing shared code. Others can be added for other platforms but, in the context of this project, only the three were required. The platform specific code (in gray in figure 5) contains UI behaviour, not shareable. The Shared module (in figure 5, in purple) contains all the functional code, defining the apps core behavior. This module contains three different directories: a common directory, containing code that is shareable by both platforms, and two others, containing platform specific APIs, like certain methods that can be called in shared code, but must be written differently for each platform.

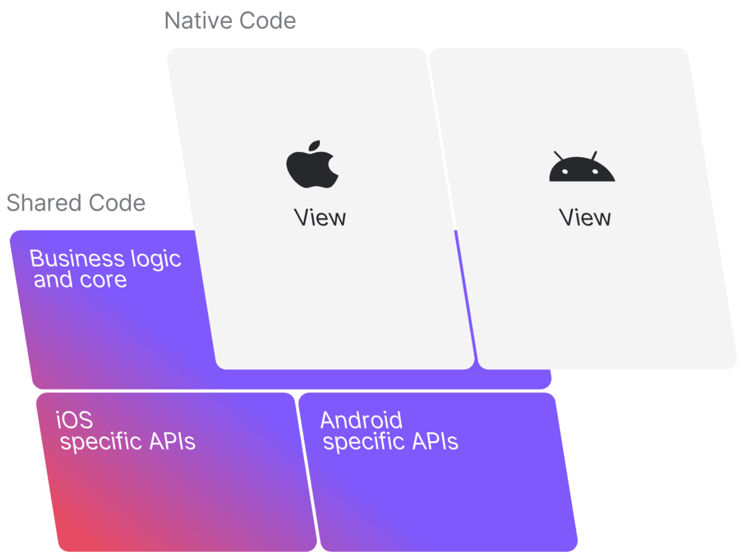


Figure 5: Kotlin's components for a Multiplatform project [12]

Despite having interoperability with Java, for multiplatform development, wrapping these methods in Kotlin classes makes it easier to write common code that both platforms can use. For example, if a Java class is needed but its behavior depends on Native platform dependent tools, that same class can be wrapped differently for both platforms, given the same name and then be gathered in a class, also with the same name, that declares its main constructor and variables, shall they exist. This class declaration is preceded by the *expect* keyword and allows it to be called in a module that both platforms can share. The classes containing the platform specific wrapper classes are preceded by the keyword *actual*. Figure 6 shows a diagram of this relation. *Actual* classes can be used, not only as wrapper classes, but to define any behavior that is platform dependent.

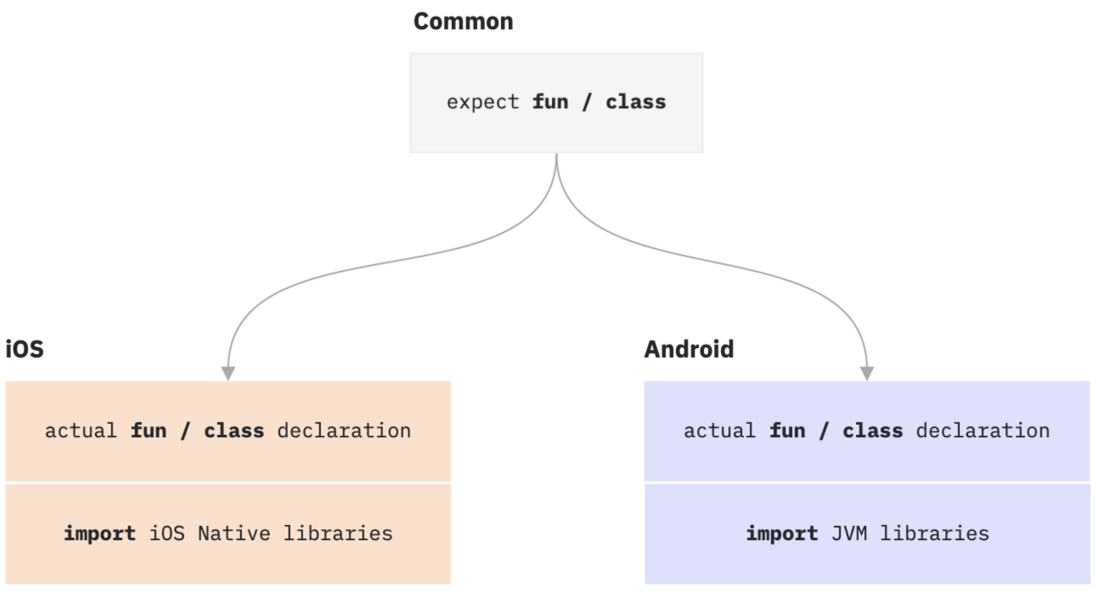

Common
expect fun / class
iOS
actual fun / class declaration
import iOS Native libraries
Android
actual fun / class declaration
import JVM libraries

Figure 6: Relation between classes with *"expect"* and *"actual"* keywords [13].

Among Kotlin's powerful tools is the Kotlin Coroutine. Even though coroutines are a known programming concept, with several programming languages providing their own version, Kotlin Courotines are ideal for mobile development. They are lightweight, have fewer memory leaks and provide built in cancellation support [14]. This allows for portions of code to be suspended when needed, canceled at anytime, providing a handler for that purpose, and a limited lifespan that depends on the context within which they are launched.

4.2.8 Gradle and Dependencies

When developing an app, for a platform, the developer will need pre-existing software, that can perform certain tasks. This will allow him to save time and implement functionalities where he does not have to be concerned over complex mechanisms working behind, that might be beyond his knowledge. This is done by including libraries in a project, and Kotlin, or any Android development kit using the JDK, does this by using Gradle. Gradle was developed in 2007 and is based on the programming language Groovy. In 2013 it was adopted by Google as the build system for Android systems [15]. Gradle is also used in Kotlin Multiplatform and allows the inclusion of tools like Cocoapods, a dependency manager for IOS libraries.

However, Gradle's function in a project is beyond including libraries. Gradle is also responsible for compilation, deployment and testing. Gradle is a build automation system which constitutes the main component for developing a mobile project and understanding its inner workings is important. For the sake of this project, and knowing that the Gradle files for the sample app being used already exist and support the apps main functionality, it is important to understand two main Gradle assets. The first, is how Gradle handles dependencies. The second concerns how the multiple Gradle files in a project interact with each other, and their function within each module.

Dependency Management Any modern software project rarely works in isolation. There is often the need to incorporate certain libraries that provide specific functionalities. Gradle provides access to several repositories, which can be seen in figure 7. Also seen in figure 7 are the specific shorthand notations used to access those repositories.

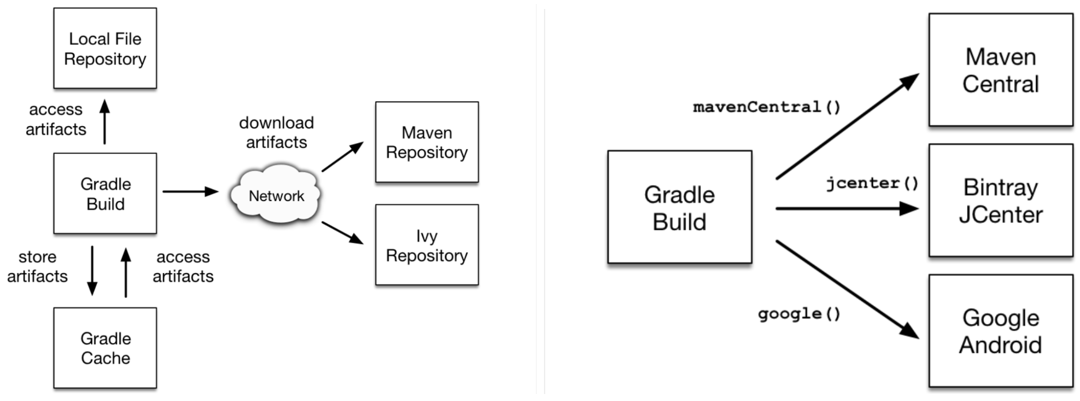


Figure 7: Different Repositories and how Gradle accesses them

Gradle Files A Kotlin Multiplatform contains several gradle files. The Android and Shared modules have their own. The Android gradle file contains information about needed libraries for anything done in this module. As mentioned it contains mostly UI behavior. It also has a path for the main implementation of this project, the Shared module. IOS does not use gradle files. Instead, it resorts to a different dependency manager called Podfile. It supports the same functions as gradle files, but for IOS code, written in Swift, and plays the same role for the IOS module as the grade file does for the Android module. The Shared module contains the most important gradle file. All the relevant imported libraries have a path and it is written in this file. It contains a section for Podfile support, Cocoapods. It also allows the developer to include different libraries, if required for each of the different platform specific APIs.

4.2.9 WebRTC

WebRTC is a free and open-source project that provides different platforms and devices with real-time communication capabilities, be it visual, audiovisual and the sending of generic data between users. *"The technology is available on all modern browsers as well as on native clients for all major platforms"* [16].

By providing simple APIs it eliminates the need for the user to install plugins or download native apps. As for mobile users, the official WebRTC page ([16]) has official Android and IOS APIs, for developers to use in the development of libraries and apps for WebRTC applications. Despite being described as relatively simple, the WebRTC mechanism still requires some understanding.

ICE (Interactive Connectivity Establishment) WebRTC advantageously works mainly through P2P (Peer to Peer) technology, meaning direct communication between the two entities and streams with as little delay as possible. This is made possible by WebRTC's implementation of the ICE (Interactive Connectivity Establishment) framework. ICE, as defined in [17] is a "technique for NAT (Network Address Translation) traversal for UDP-based (User Datagram Protocol) data streams", that allows two peers to connect to each other, regardless of network topology or whether or not they are behind NAT. To do this ICE uses ICE servers to either let a peer know the other's public IP (Internet Protocol) address, or to use as relay servers.

This framework's algorithm looks for the lowest latency path between both peers. There are several options that ICE explores in this order of priority [18]:

- **1.** Direct UDP connection
- **2.** Direct TCP (Transmission Control Protocol) connection, using the HTTP port
- **3.** Direct TCP connection, using the HTTPS (HyperText Transfer Protocol Secure) port
- **4.** Direct connection via a TURN server

STUN servers When a request is sent to a STUN (Session Traversal Utilities for NAT) server, its function is to return a string containing the type of NAT the user making the request is behind and its public address and NAT port, or too ask the remote peer for its address in order to provide a **srflx** candidate to the peer making the request. In WebRTC they are used to establish a UDP connection. Therefore, this servers are very simple and easy to maintain. As a downside, STUN servers are incompatible with Symmetric NAT. This happens because internal port mapping, inside a router running Symmetric NAT, can change and the same port can be used to send packets from different users to different web services. This means STUN servers can not guarantee the public IP address is associated with the private IP of the user making the request.

TURN servers When peers are behind symmetric NAT, STUN servers will not be able to provide trustworthy information and WebRTC will fail to start a session. There are also other reasons why WebRTC might fail. Peers might not be able to find a path they can both agree on, despite knowing all the necessary details. TURN (Traversal Using Relays around NAT) servers are a last case scenario, when all else fails. A TURN is a media relay/proxy server that allows peers to exchange UDP or TCP media traffic whenever one or both parties are behind NAT.

They also account for the technology running behind services like Skype. In order to access a TURN server the caller, responsible for making the call, sends an *Allocate* request. This request asks the TURN server to allocate resources so it can contact a peer. If possible, the TURN server will let the caller know that the resources have been allocated. The caller then sends a *Create-Permissions* request, so the TURN server can verify if the peer-to-TURN communication is valid. TURN servers offer two means of communication: the Send Mechanism, which is more straightforward but carries a 36 byte header, or the Channel Bind method, which has a small header of 4 bytes but requires the reservation of a channel, requiring constant updating. Having understood the way they operate we can understand why TURN servers are more expensive and harder to maintain when compared to STUN servers, meaning less options are available. The process just described happens in the background, and the WebRTC API is responsible for managing all these steps. However, the user must provide functional TURN servers URLs (Uniform Resource Locators) and valid credentials. Figure 8 provides a diagram showcasing the differences between communication using P2P and TURN.

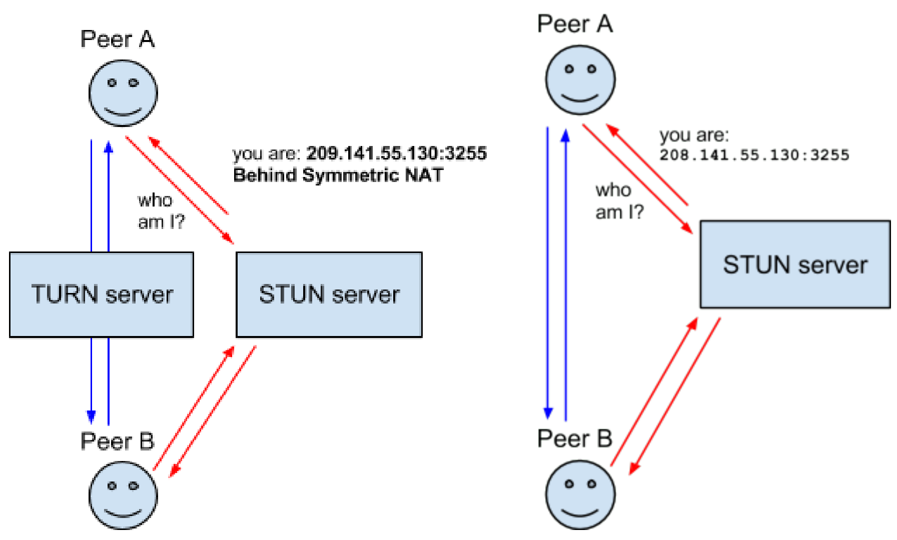


Figure 8: Differences between a connection with only STUN, and using STUN and TURN servers [18].

Ice Agent and Ice Candidates An Ice Candidate describes available methods for communication between both peers. This includes protocols and routing options. This process happens in the Ice Layer, and the Ice Agent, in each peer, is responsible for gathering and negotiating Ice Candidates. A peer will start off by sending its best candidates, making its way to the worst. The list ate the beginning of this section shows the ideal order in which the candidates are sent.

UDP candidates, being the more suitable way of establishing multimedia communications, are the first ones to be sent. A UDP candidate can be categorized as one of four types [18]:

- **host:** when the IP address contained in the candidate is the actual IP address of the remote peer. This is the preferred type of candidate, allowing direct and fast communication;
- **prflx:** short for peer reflexive candidate, identifies a peer behind symmetrical NAT. Is typically sent when trickle ICE is enabled and before connection verification is finished, meaning this candidate is unsuitable for P2P communication but the WebRTC native library has no way of verifying that yet;
- **srflx:** a server reflexive candidate is aimed at discovering the address a NAT has given the remote peer by asking a STUN server. The STUN server will return the NAT created IP address and the proper bindings will be established. The type of NAT the peers sit behind will determine the success of this type of candidate (symmetrical NATs will make these candidates invalid) [19];
- **relay:** a relay candidate will be generated similarly to **srflx** candidates, but using a TURN instead of a STUN.

TCP candidates are only used when UDP options are unavailable. They carry a heavier load, demanding more from the network, are slower and might not be the best option for media streaming, as some browsers might not support it. TCP candidates are divided into three candidates [19]:

- **active:** the ICE Agent will attempt to open a connection to the other peer but will not receive incoming connection attempts;
- **passive:** the ICE Agent will receive connection attempts, but not make its own;
- **S-O (Simultaneously Opened):** ICE Agents will attempt to open a connection at the same time.

Trickle Ice For Ice Candidate exchanging, WebRTC provides a functionality called "Trickle Ice". It allows for candidates to be sent as soon as they are collected. Traditionally candidates would be collected and then prioritized [20]. They would be put in a list and sent all at once. However this process is slow and can delay the connection. This way the Ice Agent starts sending the Ice Candidates as soon as they are gathered. Since it starts gathering the supposed best candidates first, it still prioritizes them. Better candidates can be collected later but WebRTC allows the connection to change paths if a better option is discovered.

Signaling Server Initially, and so that both peers can connect, the use of a third party is necessary. This third party is referred to as a Signaling Server and can be whatever technology the developer chooses, as long as he is able to correctly merge it with the WebRTC API. Most WebRTC samples found online use simple Websocket server and client implementations that allow for simple and fast information exchange. WebRTC is supported by most browsers and, for signaling, any JavaScript Websocket library, like its native *ws* library or *Socket.io*, are implemented alongside the WebRTC API. For Android and IOS, Websockets are also the easiest and most developer friendly option as well.

A WebRTC call happens in 3 stages: establishing initial contact, negotiation and ending the call.

1 - Establishing Initial Contact For the sake of this explanation, the local peer will be considered the one establishing the call and the remote peer will be the one receiving the call. In a real scenario, this role is interchangeable. In order to initiate a WebRTC session and establish contact, the local peer must first create an object of the type *RTCPeerConnection().* This object will represent the connection between the local peer and the remote peer. It also provides several methods to make, maintain and monitor the connection as well as close it when the call is over. After, it must collect information regarding its own video and audio feeds, by accessing the camera and microphone. It uses the method *getUserMedia().* If no camera and microphone are available the call proceeds naturally, but without a stream from the local peer. The method *addTrack()* displays the other peer's track on the screen.

The way the WebRTC API communicates with its background native library and processes is by using events, which it triggers when certain methods are called. The WebRTC native library, as a response, will also send its own events, triggering states where methods are implemented and called. At this stage of the calling process, *RTCPeerConnection()* will have triggered certain methods and events in the native library. In response the native library sends an event:

handleNegotiationNeededEvent. This will trigger the app to create a **SDP (Session Description Protocol)**, a string containing a description of the peers relevant characteristics for this session. That is done using the method *createOffer()*. The local peer also needs to know its own description. Therefore it uses the recently created SDP (Session Description Protocol) as input for the method *setLocalDescription()*. Immediately after, the SDP file is sent to the remote peer through the Signaling Server. This is known as the *"offer"*.

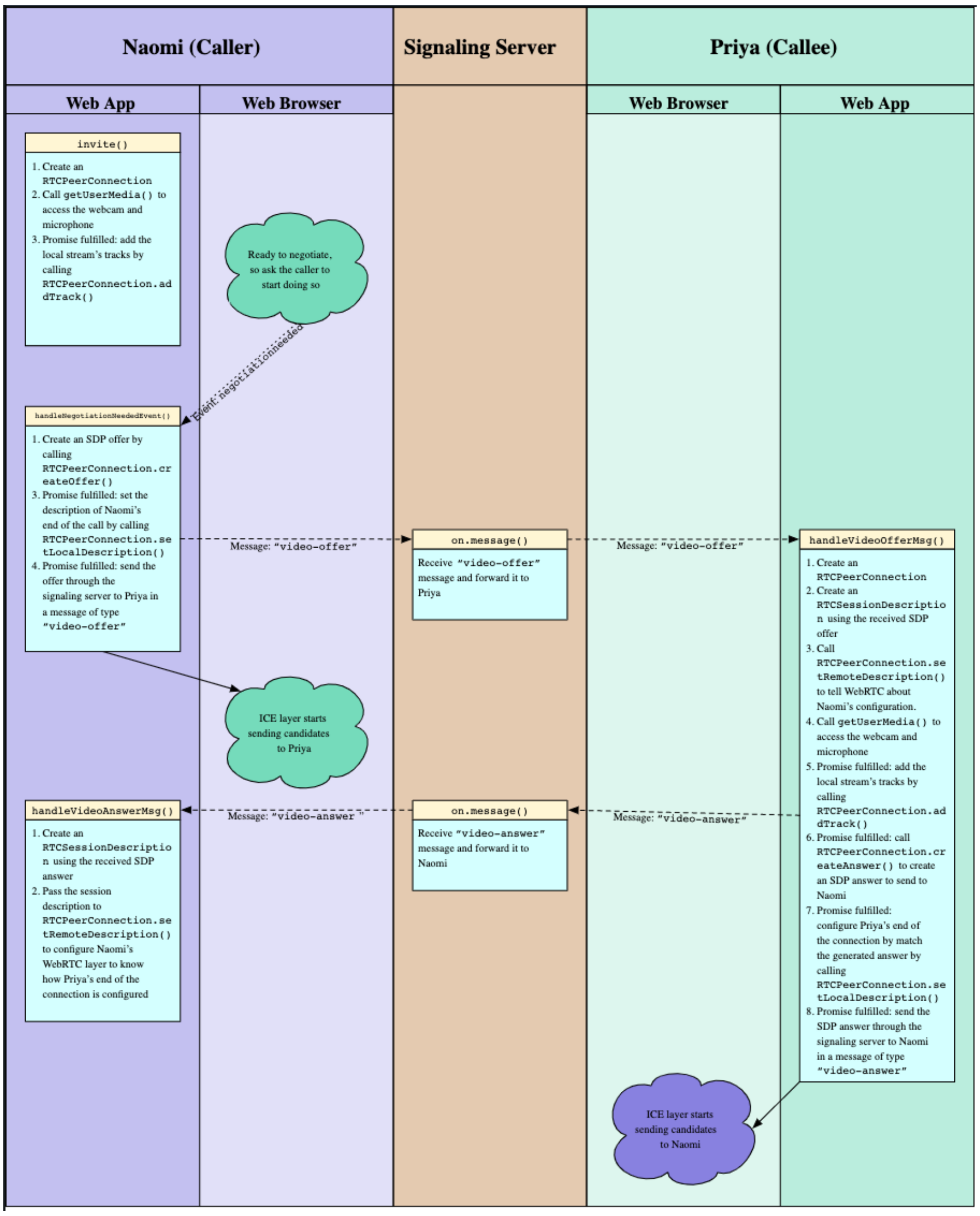

Naomi (Caller)
Signaling Server
Priya (Callee)
Web App
Web Browser
Web Browser
Web App
invite()
1. Create an RTCPeerConnection
2. Call getUserMedia() to access the webcam and microphone
3. Promise fulfilled: add the local stream's tracks by calling RTCPeerConnection.addTrack()
Ready to negotiate, so ask the caller to start doing so
Event: negotiationneeded
handleNegotiationNeededEvent()
1. Create an SDP offer by calling RTCPeerConnection.createOffer()
3. Promise fulfilled: set the description of Naomi's end of the call by calling RTCPeerConnection.setLocalDescription()
4. Promise fulfilled: send the offer through the signaling server to Priya in a message of type "video-offer"
Message: "video-offer"
on.message()
Receive "video-offer" message and forward it to Priya
Message: "video-offer"
handleVideoOfferMsg()
1. Create an RTCPeerConnection
2. Create an RTCSessionDescription using the received SDP offer
3. Call RTCPeerConnection.setRemoteDescription() to tell WebRTC about Naomi's configuration.
4. Call getUserMedia() to access the webcam and microphone
5. Promise fulfilled: add the local stream's tracks by calling RTCPeerConnection.addTrack()
6. Promise fulfilled: call RTCPeerConnection.createAnswer() to create an SDP answer to send to Naomi
7. Promise fulfilled: configure Priya's end of the connection by match the generated answer by calling RTCPeerConnection.setLocalDescription()
8. Promise fulfilled: send the SDP answer through the signaling server to Naomi in a message of type "video-answer"
ICE layer starts sending candidates to Priya
handleVideoAnswerMsg()
1. Create an RTCSessionDescription using the received SDP answer
2. Pass the session description to RTCPeerConnection.setRemoteDescription() to configure Naomi's WebRTC layer to know how Priya's end of the connection is configured
Message: "video-answer"
on.message()
Receive "video-answer" message and forward it to Naomi
Message: "video-answer"
ICE layer starts sending candidates to Naomi

Figure 9: Sequence of events too establish initial contact [16].

When the *"offer"* is received, the remote peer will have already created its own *RTCPeerConnection()* by then. With the SDP from the local peer, the remote peer will use the method *setRemoteDescription()*. This will make the remote peer aware of the local peer's characteristics. Having done that, the remote peer repeats the process described earlier using the methods *getUserMedia()*, for local media gathering, and *createAnswer()*, which behaves similarly to *createOffer()*, but contains some information regarding the session at hand. The remote peer will set his own description using *setLocalDescription()* with the SDP from *createAnswer()*, and send it to the local peer, which will call the method *setRemoteDescription()* using the *"answer"* it just received. This stage is the basis for the negotiation stage.

Figure 9 gives an overview of the process just described for two peers using common Web Browsers. It should be noted that for Android and IOS, as well as other platforms, the process is the same, but the WebRTC native library replaces the Web Browser.

2 - Negotiation The negotiation stage focuses on trying to find the best ICE candidates for communication. After being collected by each peer's Ice Agent they are sent, via the Signaling Server, to the other peer. This process is not synchronous, meaning the peer does not wait for a reply after sending an Ice Candidate. Because of "Trickle ICE", they will be sent as soon as they are collected and a peer will be doing so right after the initial offer (or answer depending on which peer started the call) is sent. Whenever an Ice Agent collects a new Ice candidate, it will trigger the *onIceCandidate* event. Using the proper event handler, whenever this event is triggered, the new Ice candidate will be sent through the Signaling Server to the other peer. The newly received Ice Candidate will be added to the *RTCPeerConnection()* object with the method *addIceCandidate()*.

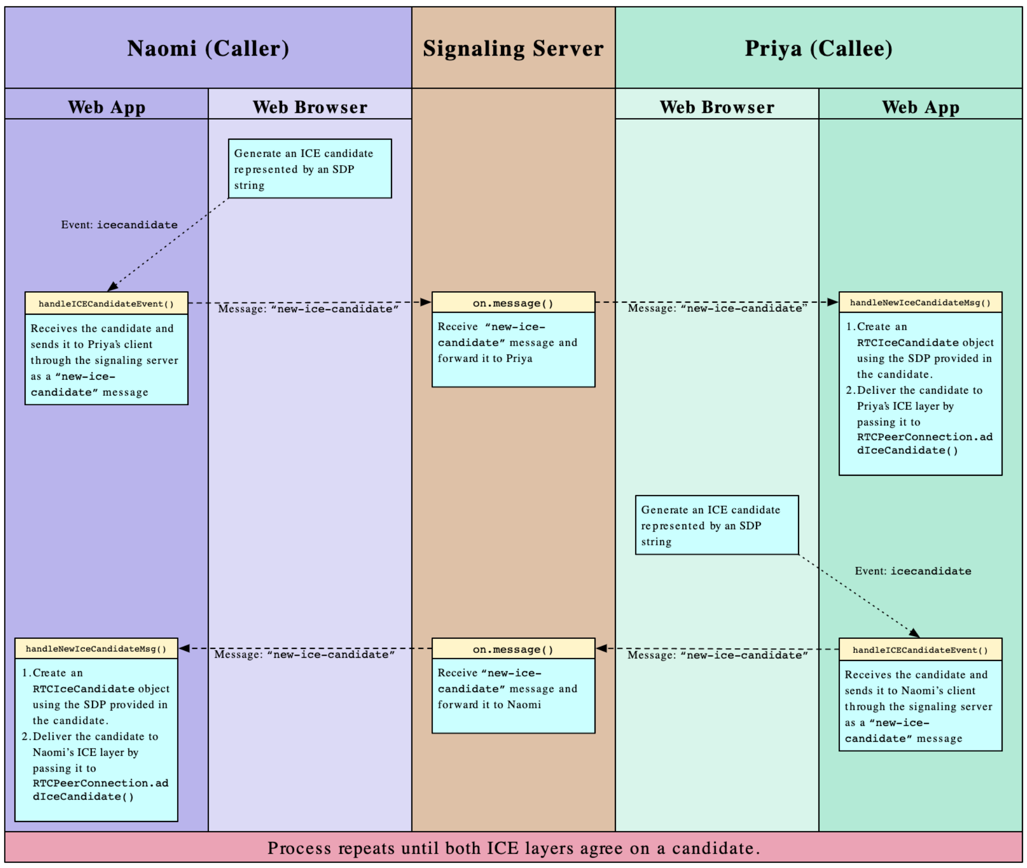

Naomi (Caller)
Signaling Server
Priya (Callee)
Web App
Web Browser
Web Browser
Web App
Generate an ICE candidate represented by an SDP string
Event: icecandidate
handleICECandidateEvent()
Receives the candidate and sends it to Priya's client through the signaling server as a "new-ice-candidate" message
Message: "new-ice-candidate"
on.message()
Receive "new-ice-candidate" message and forward it to Priya
Message: "new-ice-candidate"
handleNewIceCandidateMsg()
1. Create an RTCIceCandidate object using the SDP provided in the candidate.
2. Deliver the candidate to Priya's ICE layer by passing it to RTCPeerConnection.addIceCandidate()
Generate an ICE candidate represented by an SDP string
Event: icecandidate
handleNewIceCandidateMsg()
1. Create an RTCIceCandidate object using the SDP provided in the candidate.
2. Deliver the candidate to Naomi's ICE layer by passing it to RTCPeerConnection.addIceCandidate()
Message: "new-ice-candidate"
on.message()
Receive "new-ice-candidate" message and forward it to Naomi
Message: "new-ice-candidate"
handleICECandidateEvent()
Receives the candidate and sends it to Naomi's client through the signaling server as a "new-ice-candidate" message
Process repeats until both ICE layers agree on a candidate.

Figure 10: Peers exchanging ICE Candidates [16]

One of the Ice Agents will be given the role of *controlling Agent*, and will be responsible for choosing the best candidate pair for communication. A candidate pair is the combination of two candidates, one from each peer, that provide a matching path for communication. The other Ice Candidate will be the *controlled Agent* and will receive that information through the ICE Server. The process of choosing the controlling and controlled agents is determined by a set of rules described in [17]. The type of Ice Agent, and whether or not they are associated with the peer that established first contact are the two main factors. This offers a little more insight over the mechanisms behind WebRTC, but ultimately, for the purposes of establishing communication, it does not matter which Agent is which.

After a viable pair is chosen media can begin to flow between the two peers. This can happen at any time during the Ice candidate gathering and does not mean peers will stop exchanging candidates. Better ones might be collected by each peer and, as mentioned before, WebRTC allows the communication path to change during a session. Figure 10 provides an overview of the candidate exchange process.

4.2.10 WebRTC as a state machine

When implementing and debugging a WebRTC app, it is important to understand how the front-end API communicates with the back-end library. This is accomplished by the exchanging of events. These events trigger different states, that provide information about the correct (or incorrect) behavior of the app. Checking for mismatched states is very important to build a reliable app. A WebRTC app has four standout properties and two interfaces, all of them read-only, with several states each, described below.

The RTCIceTransport and RTCDtlsTransport interfaces Before mentioning WebRTC's most important properties it is important to first quickly mention the RTCIceTransport and RTCDtlsTransport interfaces. Both interfaces provide information about the transport layer of the WebRTC library. They are state machines and their state affects how the properties behave. They also provide some of the same information. For instance, the current state of the Ice Connection State and Ice Gathering State properties can both be obtained by reading the **state** and **gatheringState** porperties of RTCIceTransport, respectively. Most importantly, they provide information about how the Ice Agent sends and receives candidates, as well as its role in the communication.

Signaling State Despite what the name might suggest, these state is not directly influenced by whether the Signaling Server is or not connected. This should be emphasized, since, as mentioned before, the Signaling Server is implemented separately from the WebRTC API. However, the implementation of certain API methods is influenced by when the Signaling Server receives a message, and what type of message that is. Therefore, we can say this property is indirectly influenced by the messages received via the Signaling Server. The six possible states below describe how this happens and figure 11 provides a better understanding of how they interact.

- **stable:** this state can have two meanings. On one hand, it can mean the *RTCPeerConnection* is new and no offer has been made. It can also mean the *"offer"* and *"answer"* exchange is complete and either the negotiation is underway or a session has been established.

- **have-local-offer:** after calling *createOffer()* and using the resulting SDP as input for *setLocalDescription()*, this state is triggered and the *"offer"* can be sent to the remote peer. Only the local peer will be in this state.
- **have-remote-offer:** the remote peer will receive an *"offer"* through the signaling server and, by calling the method *setRemoteDescription()*, will trigger this state.
- **have-local-pranswer:** after receiving an *"offer"* and setting the remote description, the remote peer will create its own *"answer"*, via the method *createAnswer()*, and set its own local description using *setLocalDescription()*. The *"answer"* will be sent via the Signaling Server.
- **have-remote-pranswer:** once the local peer receives an *"answer"* it will set the remote description with *setRemoteDescription()* and trigger this state.
- **closed:** the session has been closed.

It is important to mention that **have-local-pranswer** can only be triggered if the previous state was **have-remote-offer** and the same is true for **have-remote-pranswer** and **have-local-offer**. Equally important, and mentioned before, the local peer can begin gathering candidates right after *"RTCPeerConnection.setLocalDescription()"* and receiving them before *"RTCPeerConnection.setRemoteDescription()"*. WebRTC does not know how to deal with candidates if descriptions have not been set yet. This is justified because WebRTC, in attempts to try and be as efficient as possible will begin negotiations as soon as possible. If descriptions are yet to be set, the system will not know if a candidate pair is viable. Therefore, when they are being received, candidates are put in a queue and, once the **stable** state has been triggered, these candidates will be eligible for pair checking.

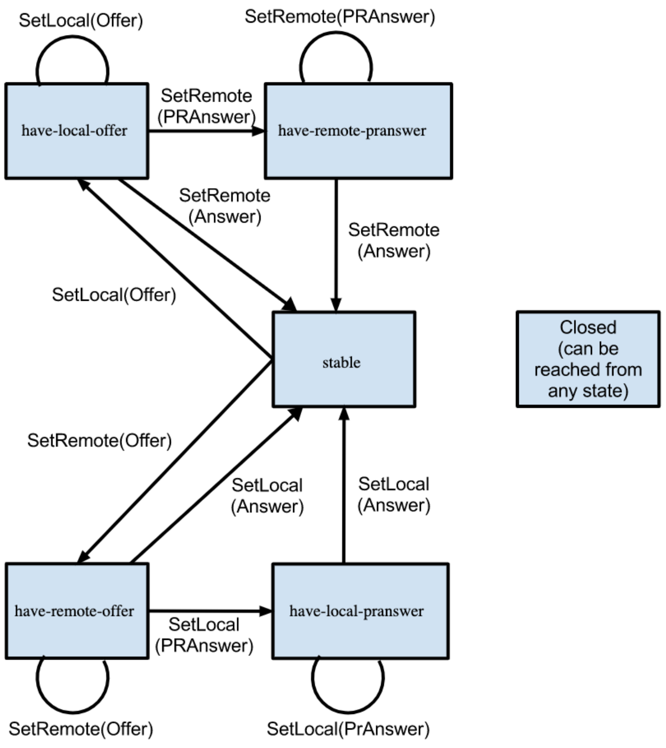


Figure 11: Signaling State transition diagram [21].

Ice Connection State This property returns a string showing which state the Ice Agent is in, regarding its connection to the Ice Server (STUN or TURN). There are 7 possible states which are described according to [21]. Figure 12 shows which states are connected and the possible direction of the transitions. Unlike figure 11, figure 12 does not show which events lead to state transitions, since this process happens without the developers interference.

- **new:** the Ice Agent is gathering local candidates or waiting to receive remote candidates from the Signaling Server. This state will change once the Ice Agent has gathered a local candidate or the *addIceCandidate()* method has been called.
- **checking:** the Ice Agent has received one or more remote candidates from the Signaling Server and is now checking for pairs of local and remote candidates, but is yet to find a viable pair.
- **connected:** a pair of candidates has been found and communication can now begin. Candidate gathering might still be underway.
- **completed:** Gathering is complete and the Ice Agent has found viable candidates. Communication might already be underway.
- **failed:** Gathering is complete and the Ice Agent was unable to find viable candidates.
- **disconnected:** if some aspect of the connection fails, this state triggers the system to look for a solution until the connection returns, triggering the **connected** state again. If no solution is found among the candidate pairs, the **failed** state is triggered and the connection cannot carry on.
- **closed:** The session has been shutdown.

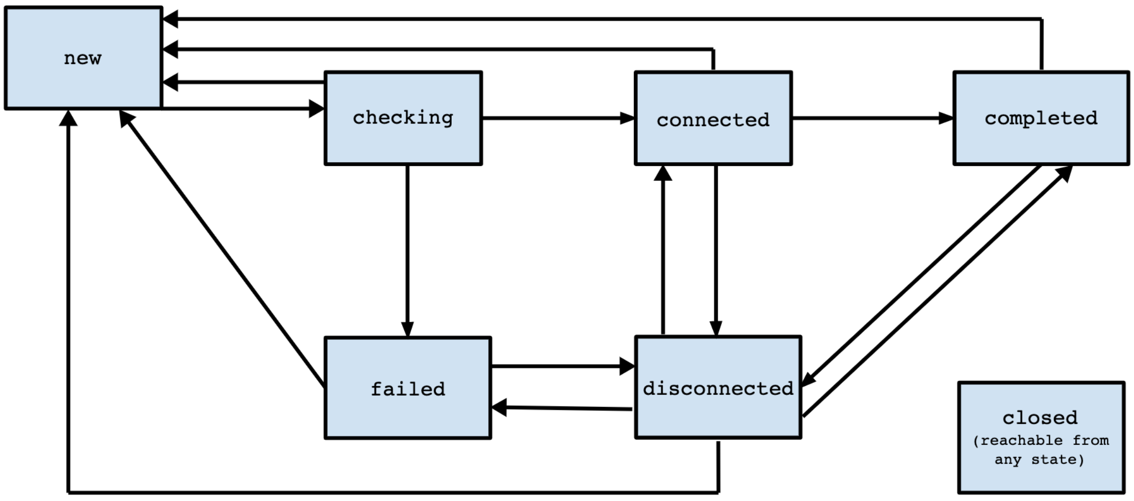


Figure 12: Ice Connection State transition diagram [21].

Ice Gathering State This property also concerns the ICE gathering of local candidates. However, unlike the previous property, the checking of candidate pairs is not relevant, nor the actual state of the connection. Instead, this property merely informs the system of the state of local candidates gathering.

This property impacts the Ice Connection State property, informing states like **new**, **completed** and **failed**. This is reflected by how the system can be gathering candidates and, at the same time, comparing candidate pairs before all local candidates are gathered, or all remote candidates are received. A clear example of how the WebRTC session is organized and how it takes advantage of asynchronous systems for efficiency.

Connection State The Connection State property, as the name implies, returns a string describing the current state of the connection. It provides an overview of the entire system, and checking for mismatched states in this property is the first step for debugging if the app malfunctions. It mainly looks at the state of both transport interfaces described earlier. The connection can be in one of six possible states:

- **new:** checks if the state of Ice Connection State is **new**.
- **connecting:** either of the RTCIceTransport or RTCDtlsTransport interfaces are in the process of establishing a connection, meaning the Ice Connection State is checking or connected. None of the interfaces are in **failed** state.
- **connected:** at least one transport is in connected or completed state.
- **disconnected:** at least one of the transports is in the **disconnected** state.
- **failed:** one or both transports are in **failed** state.
- **closed:** the session has been closed

4.2.11 SDP (Session Description Protocol)

When starting multimedia communications it is necessary to convey to the participants the details of the session. These details can include the type of media being shared, its format, transport protocols and others. The Session Description Protocol was created for that purpose. It was specified as a standard for describing multimedia sessions in 2006 [22].

The SDP is comprised of several lines, divided into three sections: session description, followed by one or more media descriptions, and a time description. Each line of text is in the form:

$< type >=< value >$

The structure of a SDP description is shown below:

Session description
v= (protocol version)
o= (originator and session identifier)
s= (session name)
i=* (session information)
u=* (URI of description)
e=* (email address)
p=* (phone number)
c=* (connection information – not required if included in all media)
b=* (zero or more bandwidth information lines) One or more time descriptions (”t=” and ”r=”

lines; see below)
z=* (time zone adjustments)
k=* (encryption key)
a=* (zero or more session attribute lines)
Zero or more media descriptions

Time description
t= (time the session is active)
r=* (zero or more repeat times)

Media description, if present
m= (media name and transport address)
i=* (media title)
c=* (connection information – optional if included at session level)
b=* (zero or more bandwidth information lines)
k=* (encryption key)
a=* (zero or more media attribute lines)

It is important to account for the lack of spacing between the <type>, equals sign ("=") and <value>. According to [22] the <type> must be exactly one character, case-sensitive, and <value> can be a number of fields, defined by the <type>, separated by a single whitespace. Some of the lines are required and others, marked with "*", optional. However, the order they appear in must be respected inside each section.

4.2.12 UV4L

When looking to implement multimedia streaming on a Raspberry Pi, the best available option is UV4L.

UV4L "offers a generic purpose Streaming Server plug-in, especially made for IoT devices, that can serve custom web applications that can leverage a number of standard and modern built-in services for Real-Time Communications such as WebRTC, allowing encrypted and bidirectional audio, video and data streaming or conferencing over the web" [9].

UV4L is a particularly powerful tool for WebRTC. It supports a server with its own built in web page. It also hosts a websocket server, acting as signaling server, with its own instructions on how to establish initial contact. The web page's source code is accessible to the user, and implements methods from the WebRTC API in JavaScript. It can act as a guide for someone trying to develop a WebRTC app for multimedia streaming from a Raspberry Pi running a UV4L server.

It also provides an instruction manual for quick installation on a Raspberry Pi and a long configuration file, giving the user a lot of options regarding security, codecs, media formats, and others.

Figure 13 shows UV4L's UI, available in any browser.

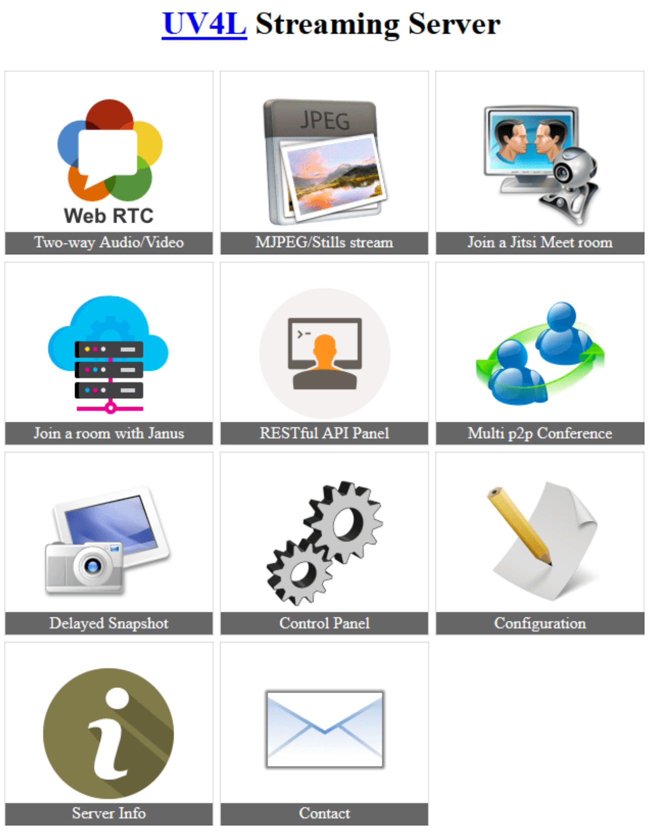


Figure 13: UV4L built in web page.

5 Methodology

The Gantt map in figure 14 provides an overall view of the stages this project went through.

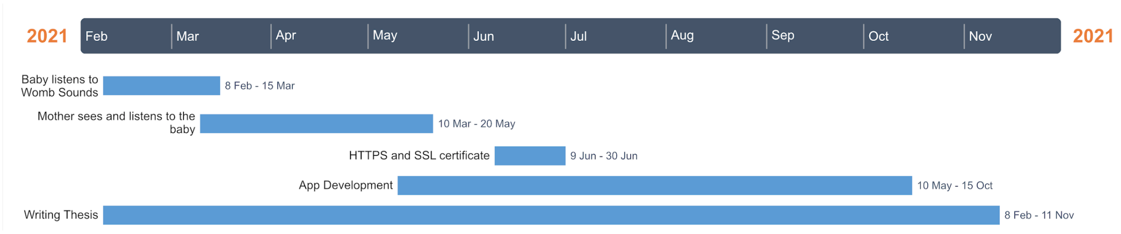


Figure 14: Gantt Map

Baby listens to Womb sounds At the start of this project, part of the Baby Side Implementation was already in the works. The stethoscope was sending audio to the smartphone which was sending it to the Raspberry Pi. This stage was defined by the implementation of a method capable of redirecting the audio to a server where it could be accessed by a different Raspberry Pi and then played. This consisted of installing and configuring the PulseAudio, DarkIce and IceCast modules.

Mother sees and listens to the baby On a second stage, work was focused on assembling a mechanism that could collect audio and video synchronously, package it and send via the internet on a Raspberry Pi. During this stage, UV4L was implemented in the Raspberry Pi assembled next to the incubator. Our hardware choices were tested at this stage, after the software implementation. This allowed checking for eventual incompatibilities between UV4L and the available camera and microphone.

App Development This stage was started as soon as the stream for the incubator was up and running. UV4L runs WebRTC, which meant developing an app that could support WebRTC in a mobile multiplatform environment. As stated before, the chosen framework for this work was Kotlin Multiplatform. A large portion of this stage was dedicated to learning this framework, as well as understanding WebRTC and how to implement it in a mobile context. This app is also ran on a different smartphone from the one from module 1, figure 1 from chapter 3. This is due to the fact that the Stemoscope is dependent on an app whose functionality conflicts with the app we are developing.

HTTPS and SSL certificate Due to constraints in the way WebRTC and the Android platform work, this stage was dedicated to understanding some of the connection limitations between the Raspberry Pi and the device in which the app was being tested. This limitations were due to valid certification in the server ran by UV4L and access to a CA (certificate Authority) was needed.

Writing Thesis Part of the work done throughout this project was focused on writing this document.

5.1 Risk Analysis

When preparing this project, the main risks considered were related with equipment. Looking back, certain software limitations should have been taken into account so that proper mitigation strategies could have been discussed.

5.1.1 Main Risks

With all the information laid out so far, we assumed that, in an initial phase, the main risks with developing this project were going to be related to equipment limitations. The use of a smartphone to send the sound, collected by the stethoscope, and also receive a multimedia stream could lead to problems regarding audio handling. A smartphone might send both audios, the baby's and the mother's, at the same time to the Raspberry Pi, and the baby hear its own audio along with the womb sounds.

Another risk was related to the fact that the speakers described in the previous chapter are optimized to play sound through womb walls. The amniotic liquid carries the sound to the baby so he can listen to music. Not only is the incubator's surface a very different material from the mother's belly, but the sound does not not propagate as easily through the air inside the incubator as it would through amniotic liquid.

The microphone could reveal itself to be a problem as well. Being a contact microphone, it is very sensitive to any type of contact made with whatever surface it is in contact with. This may result in a very noise recording of the baby.

It was also important to consider that it might be difficult to have bidirectional streams fully functional inside a Raspberry Pi and then have those streams accessible in multiple devices. This is due to these systems' incorporated methods for dealing with audio and video. Raspberry Pis use sinks as destination for audio so that third party apps can access them and use the audio. This leads to conflicts between those apps because they might try to access these sinks at the same time. Also, a sink needs a source of sound and multiple streams means multiple sources of sound.

Besides all the technological risks already discussed, it was important to mention that, given the pandemic situation, implementation and testing could be difficult to carry out. Finding a incubator in which the prototype can be tested might not be possible.

5.1.2 Mitigation Strategies

It was important to plan out early solutions for the main risks laid out earlier.

The issues relating to the speakers could be solved by increasing its gain. However, that might not work due to what the speakers are designed for. If that were to be the case different speakers would have to be considered, maybe something more invasive or with a different purpose that still works through walls.

When dealing with the Raspberry Pis' sound management, there are methods for controlling sink access by multiple apps. Also, the Raspberry Pi has different sinks, each with a different purpose.

In the long run, other solutions could be analyzed to facilitate the use of this prototype, regarding the smartphones and the Raspberry Pi. Since the Stemoscope depends on the developer's app, some reverse engineering, or other methods, could be considered to allow more freedom when working with these devices. This would lead to the Raspberry Pi on the mother's side no longer being required. Not only that, but the developer was also looking to launch a similar device to the Stemoscope, but with the freedom to be used as a regular microphone, which would allow this system to work with only a smartphone or a Raspberry Pi. Both of these approaches could lead to a more comfortable and easy use of the prototype.

The microphone's behavior could be improved using noise control techniques and sound engineering methods. If the noise revealed itself to be overwhelming, similarly to the speakers, different equipment would have to be looked for.

5.2 Success Criteria

The initial criteria for success is dependent on the functionality of the prototype. Both streams in both directions must be operational within the constraints of the Functional and Non-Functional Requirements described in chapter 3. The mother's womb sounds must be streamed for long periods of time and heard in the incubator. This requires the correct connection between all the modules described in chapter 3.

The stream containing the baby's video and audio must be collected, synchronized and sent using UV4L. It must also be observable in the app. Success at this stage can be divided in two portions. First, is the stream being sent and is it available in a remote device? UV4L hosts a web page where the stream can be observed and its quality evaluated, which will dictate our success criteria at this stage.

Second, is the app fully functional? Implementing the app comprises several stages, which can have their own success criteria. Initial contact with the server must be correctly established in a stable manner. The negotiation stage must be correctly set up, so the P2P call can happen. On a final stage, success will be defined by whether or not the stream is available in the app, without interruptions and whenever the mother wishes to access it.

6 Implementation

6.1 Description of the Prototype

Figure 15 shows the prototype fully assembled. The Raspberry Pi to the right is the one that will accompany the mother and send her audio to the incubator. This Raspberry Pi runs the PulseAudio, DarkIce and IceCast modules. Next to it is the smartphone running the Stemoscope app. The Stemoscope is the device located above it. All this devices will connect via BLE and IceCast will be providing the audio online.

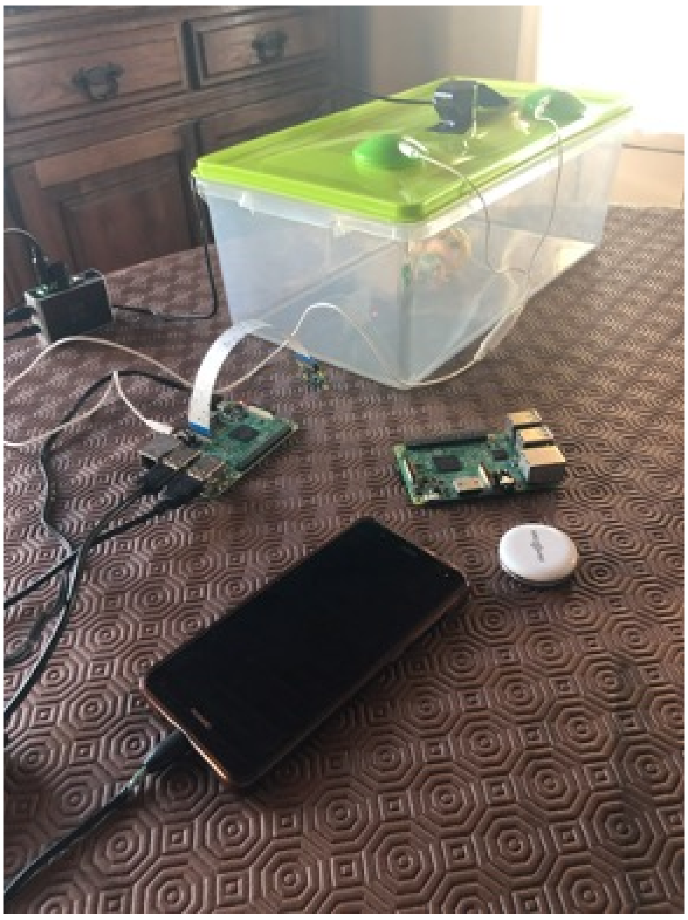


Figure 15: Fully assembled prototype

The Raspberry Pi to the left is assembled next to the incubator, represented by a Tupperware, and, as we can see, has multiple devices connected to it. The Raspicam is at the end of a strip, above the Raspberry Pi, and directed at the incubator. Above the incubator are the speakers, the green devices, also connected to the Raspberry Pi; and the contact microphone, which is connected to an amp, to the left, responsible for providing gain, which is connected and powered by the Raspberry Pi. This Raspberry Pi is running VLC, for providing the womb sounds, through the speakers, to the baby and UV4L, for collecting, synchronizing and sending the video and audio stream.

6.2 Hosting Servers on a Raspberry Pi

For this project two servers are set up for establishing communication, one running on each Raspberry Pi. For that, two public ports in the router are redirected to each Raspberry Pi. The Raspberry Pi that accompanies the mother is running the IceCast server, which is supported by standard HTTP.

The Raspberry Pi by the incubator is running a WebRTC server, supported by UV4L. UV4L runs on HTTPS. A custom URL was also created using the service No-IP (explain No-IP): "https://incubadoraiot.hopt o.org:80/stream/webrtc".

6.3 Baby listens to Womb sounds

Initial set up of the Stemoscope consisted of connecting it, via BLE, to a smartphone running the Stemoscope app. The Smartphone sends the womb sounds to the Raspberry Pi, also via BLE.

For this prototype the Raspberry Pi that accompanies the mother (figure16) was configured to behave as a Bluetooth Speaker and, at the same time, host a server. The material for this side of the prototype is in figure 16.

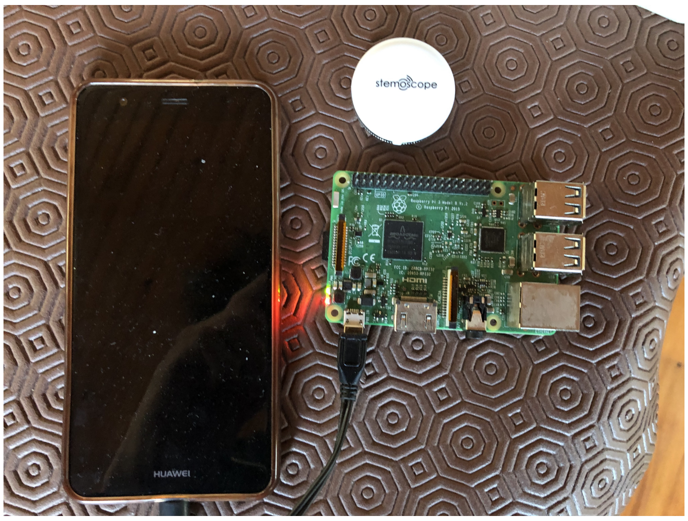


Figure 16: Material for Mother Side Implementation of the Prototype

Setting up the Raspberry Pi as a Bluetooth Speaker requires the installation of the PulseAudio Bluetooth module. This module is connected to the audio sink service, to which it will redirected the audio, after it is received from the smartphone. That is done by editing the *.rules* file, containing a set of rules defining the Bluetooth module's way of interacting with ALSA. PulseAudio will then collect this audio and send it to one of ALSA's sound card. PulseAudio is using the one associated with the Raspberry Pi's jack output. A complex shell script is implemented and its purpose is to save in a file all devices paired with the Raspberry Pi and offer audio routing options. Bluetooth pairing is done traditionally, through the smartphone.

DarkIce is responsible for collecting the sound from the Sound Card and encoding it. The Sound Card is specified in its configuration file. In this file is where the designated server (IceCast) is specified, as well as the port and local IP address to be associated with the public port on the router.

IceCast, the framework providing the server services for audio streaming, provides a m3u file that can be reproduced by several multimedia players.

VLC is the multimedia player that will play the m3u file in the Raspberry Pi by the incubator and send the womb sounds to the baby. The speakers are connected to this Raspberry Pi and are assembled on top of the incubator. This system can be seen in figure 17.

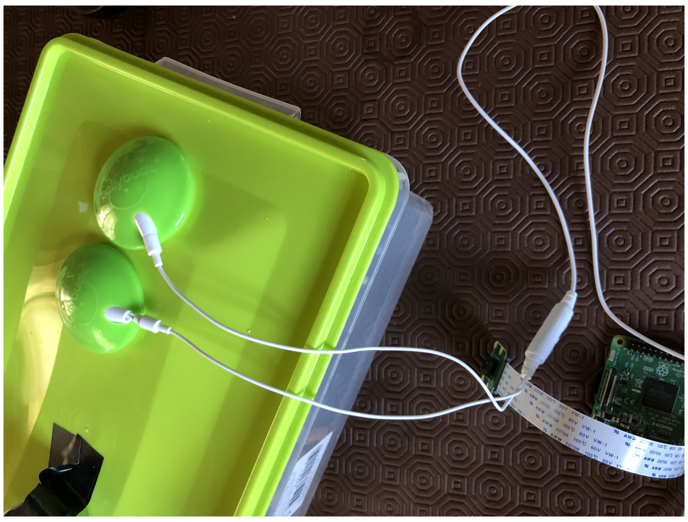


Figure 17: Implementation for playing Womb Sounds

6.4 Mother sees and listens to the baby

Besides the speakers, the Raspberry Pi next to the incubator has a Raspicam and a contact microphone connected to it. The Raspicam is directed at the baby and collects the video through the incubator's transparent walls. The contact microphone is put on top of the incubator and attached to its surface. It collects any sound wave that comes in contact with that surface. This implementation is shown in figure 18.

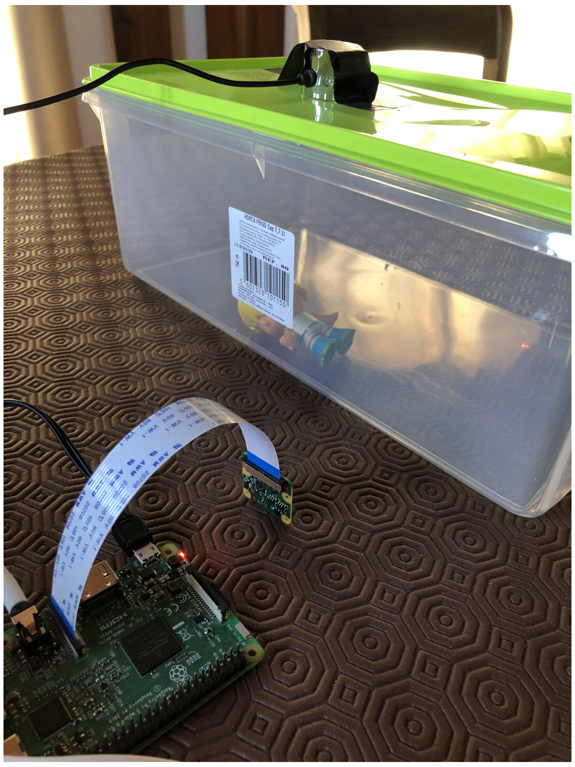
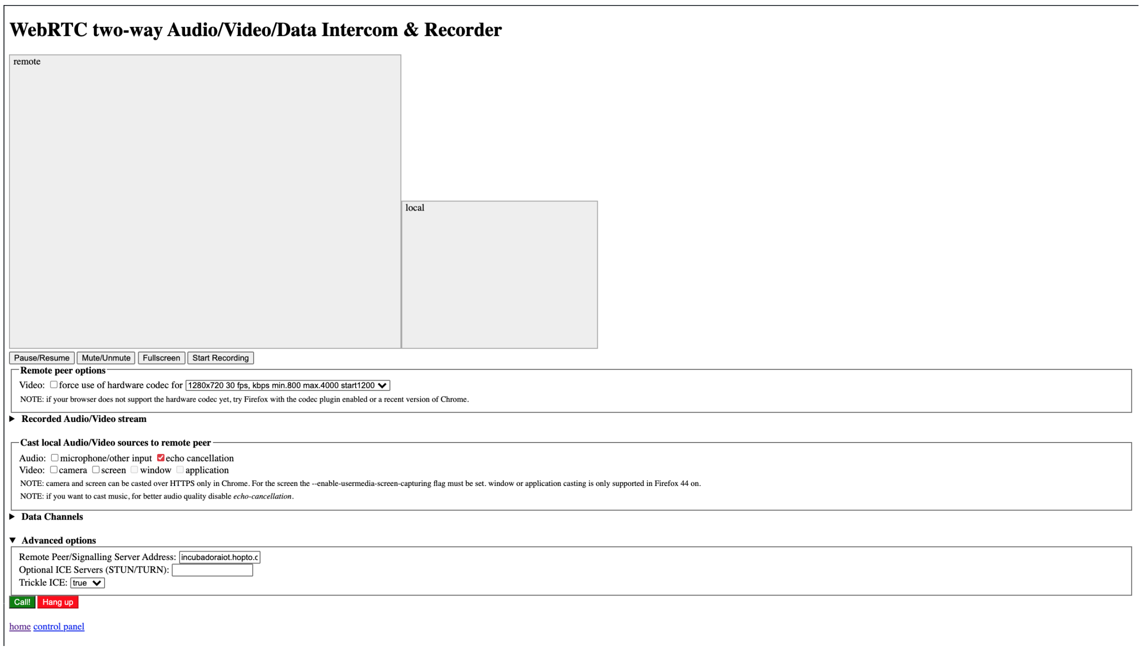


Figure 18: Implementation for collecting the Baby's video and audio

The video and audio are directed to UV4L, which is responsible for synchronization and making the stream available online, so that the mother can see her baby when she desires.

UV4L After installation on a Raspberry Pi, UV4L must be correctly set up. A set of STUN servers, and at least one TURN server must be provided in UV4L's configuration file. There are several free options available online. UV4L requires a path to the video and audio sources, which can be discovered using several Linux command tools.

WebRTC, as stated in chapter 4, is a P2P communication framework that allows two peers to communicate with each other. This means that both peers send and receive video and audio. In this prototype UV4L is only required to send video and audio. In the the UI from figure 19, where a user can establish a call and access UV4L's remote stream, the local video is displayed in the smaller window and the larger window shows the remote stream sent by UV4L. UV4L's configuration file provides the option not to receive the local stream.


WebRTC two-way Audio/Video/Data Intercom & Recorder

Remote peer options

Video: ☐ force use of hardware codec for 1280x720 30 fps, kbps min.800 max.4000 start1200

NOTE: if your browser does not support the hardware codec yet, try Firefox with the codec plugin enabled or a recent version of Chrome.

▶ **Recorded Audio/Video stream**

Cast local Audio/Video sources to remote peer

Audio: ☐ microphone/other input ☑ echo cancellation

Video: ☐ camera ☐ screen ☐ window ☐ application

NOTE: camera and screen can be casted over HTTPS only in Chrome. For the screen the --enable-usermedia-screen-capturing flag must be set. window or application casting is only supported in Firefox 44 on.

NOTE: if you want to cast music, for better audio quality disable *echo-cancellation*.

▶ **Data Channels**

▼ **Advanced options**

Remote Peer/Signalling Server Address: incubadoraiot.hopto.c

Optional ICE Servers (STUN/TURN):

Trickle ICE: true

Figure 19: UV4L built in WebRTC UI.

6.5 App and multiplatform development

6.5.1 WebRTC in Kotlin

As stated before, Kotlin has Java interoperability, meaning Java classes and methods can be imported easily into Kotlin code. However, unlike Xamarin, Kotlin requires some knowledge of the Swift programming language, which makes development for IOS harder. Therefore, initial development of the app was focused on the Android platform. An open-source library with Kotlin wrappers for the Java standard WebRTC library is available on Github, alongside with a sample app.

The Kotlin sample app also provides a loopback sample, with the necessary behavior for both the local peer and the remote peer. Its UI can be seen in figure 20. In this context, despite initiating contact with the Raspberry Pi, the smartphone will behave as the remote peer. This is due to the fact that the initial message sent to the signaling server running on the Raspberry Pi merely announces the user's smartphone's presence and its intent to start a call. The Raspberry Pi is the one to send an *"offer"*, described earlier as the first type of formal WebRTC contact between the two peers. The steps for establishing the WebRTC call between both peers in a Kotlin to UV4L context can be adapted from figures 9 and 10.

WebRTC is designed mainly as a web app. In chapter 4 it is mentioned that WebRTC is supported by most commercially available web browsers. However, there are no commercially available native apps that support WebRTC. The open source apps, available online, are meant to support communication between devices running the same app. Therefore, in order to establish a WebRTC call between UV4L and a mobile native app some extra steps were required. These include using different libraries and methods for signaling, string parsing and manipulation, specific event handling and others.

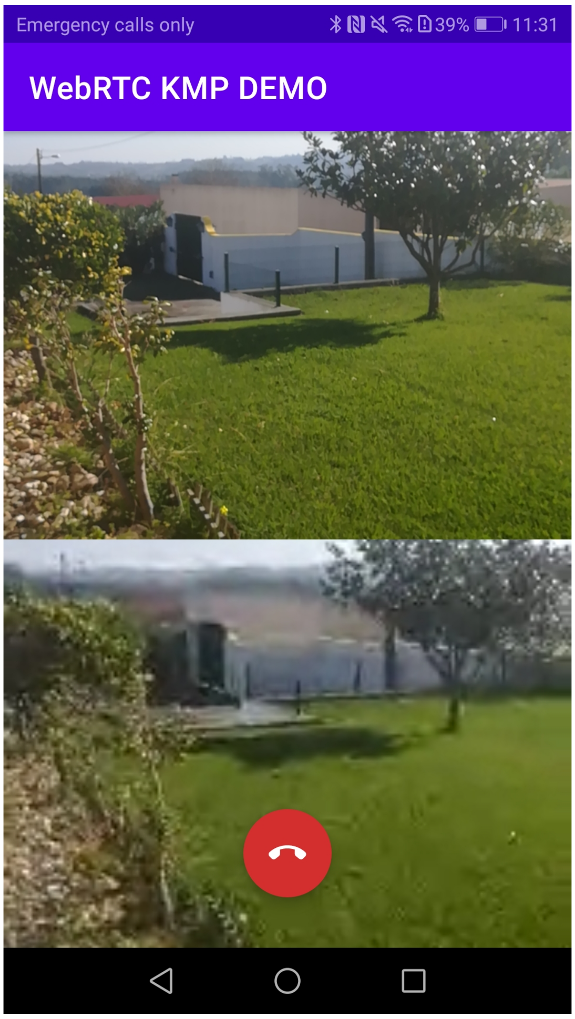


Figure 20: User Interface of the App

Kotlin's WebRTC Library WebRTC sample app's default behavior provides an implementation with both peers in the same app. It does this by creating two *RTCPeerConnection()* objects, and using the generated SDP file as an input for *setLocalDescription()* and *setRemoteDescription()*, after *createOffer()* and *createAnswer()* are called. This model also provides proper event handlers, for when any of the properties described above change state. Some of these event changes are handled with a log output, and others with specific actions, already described in chapter 4. An example of this is how the object describing the local peer would only establish a call if the *IceConnectionState* property was in the **connected** state.

In order to contact the Signaling Server a websocket library is implemented in the app. [13] provides a websocket sample app to demonstrate an application of *expect* and *actual*. It provides message *send* and *receive* methods, the latter being an event handler, triggering specific code whenever a message is received. Messages from UV4L's WebRTC implementation have different types, which will trigger different WebRTC behavior in the app.

A third library that allows classes to be *serializable* is also required. Messages coming from the signaling server carry important information for the WebRTC call (SDP and Ice Candidates). This messages come in JSON (JavaScript Object Notation) format and must be converted to objects the app can interpret.

It is important to bear in mind that different WebRTC implementations have certain small differences. When establishing initial contact each peer sends a string containing the SDP file. The format of this SDP file will depend on how the serializable classes are defined. With UV4L, its WebRTC JavaScript implementation, behind the behavior of the UI seen in figure 19, defines an object containing two strings, one named *what*, containing the type of message being received, and another called *data*, containing the relevant information. The data string can then be used as an input to the *Session Description* class constructor, generating an object which can be used as an argument for the method *setRemoteDescription()*.

In the app, a serializable class with two strings, *what* and *data*, was implemented, so that the information from the JSON message can be used.

A second serializable class was created so that JSON messages containing Ice Candidate information can be parsed into an object. It contains three fields, *candidate*, *sdpMLineIndex* and *sdpMid*. This fields contain the characteristics about the Ice Candidate described in chapter 4.

Missing Constructors WebRTC's Kotlin Multiplatform Library, has very low GitHub engagement. It is a very recent library that has not been tested by anyone outside its creator. Although very well organized, and one of the main tools to understand WebRTC's implementation, it lacked certain methods that are needed for the completion of this app and prototype.

After receiving an *"offer"* message from the Raspberry Pi, the WebRTC Kotlin Multiplatform Library did not provide a constructor for the *Session Description* class, making it impossible to use the SDP for *setRemoteDescription()*. The Kotlin Session Description wrapper classes were adapted to include a constructor.

The WebRTC Kotlin Multiplatform Library did not provide a constructor for the Ice Candidate class. This constructor required implementation as well in order to use the information from the second *serializable* class.

String Parsing Methods for string parsing were required so that the SDP string from the first *serializable* class could be adapted to our Session Description constructor. It mainly deals with $\backslash r \backslash n$ characters present in the SDP string, so that errors like *Session Description is NULL* can be avoided. This error prevents the creation of an *answer*, since details from the other peer can not be given to the system.

Sample App's UI UV4L's visual implementation of WebRTC is also different from Native apps. WebRTC's library for mobile uses the *Surface View Renderer* class, which defines its own dimensions for the stream, specific to each mobile device. This is another reason why WebRTC is not prepared for communication between different types of implementations, since *Surface View Renderer* is not compatible with the stream coming from UV4L.

WebRTC's sample app does not provide a wrapper for *Surface View Renderer*, since it is designed for communication between mobile devices. It was not possible to create the adequate class wrappers in time for the deadline for this project.

6.5.2 Signaling Server, HTTPS and SSL (Secure Sockets Layer)

For security reasons, most browsers are not compatible with WebRTC via HTTP. The same goes for the Android System. When using a browser a message like "Your connection is not private" will prevent a user from entering a website. It deems it untrustworthy. In this work UV4L is running a WebSocket server. Using a WebSocket Test Client tool, on an Android device, if the server is not behind a safe HTTPS DNS (Domain Name System), the error "Trust anchor for certification path not found" will appear in the message log.

While it is possible to overturn security measures when accessing a web page through some mobile browsers, access to a WebSocket server or web page through a native app will be blocked by the Android System. This means having a proper SSL certificate validated by a CA is required. The installation manual of UV4L for ARM (Advanced RISC Machines) provides instructions for generating a valid certificate and password-less private key. However, since this certificate is self generated and the Raspberry Pi is not a Certificate Authority, browsers and the Android System will not trust this certificate. *Let's Encrypt* is a free CA that, as long as the user demonstrates full control over the domain, will provide a valid certificate. Using *no.ip*, a free service that allows the registration of a DNS by associating it with a public IP address, it was possible to associate a router's IP with the new DNS. The chosen Hostname was *"incubadoraiot"*. *No.ip* provides several available domains, including *"hopto.org"*. Therefore, the full DNS was *"https://incubadoraiot.hopto.org"*.

For users with access to Raspberry Pi's shell, *Let's Encrypt* uses the Certbot ACME (Automatic Certificate Management Environmen) tool. By choosing the operating system and type of server being ran, the Certbot website provides instructions on how to install Certbot and create a valid certificate and key. UV4L will need access to the certificate and key just created, which can be done in its configuration file by providing a path to their folder.

7 Conclusion and Future Work

Working on this project provided a deep dive into how IoT development requires understanding of different technologies. Building a prototype of this scale is based on the assembling of different developer tools and having them cooperate.

Choosing hardware that could be assembled around an incubator in a non-invasive way was properly researched, with time and before the project started. Mitigation strategies were also considered beforehand. However, the same can not be said about software tools necessary for the prototype's functionality. Choosing appropriate software was improvised throughout the duration of the prototype's development, leading to delays. Lack of mitigation strategies for this type of problems lead to difficulties and delays, reaching dead ends and having to go back and rethink solutions with new research.

For instance, prioritizing multiplatform development was a misstep, since Xamarin turned out to be incompatible with WebRTC and Kotlin quite limited in this context.

Analyzing our success criteria we can conclude this project was partially successful.

The implementation of the side of the prototype in which the baby listens to womb sounds was successful. The sounds are collected, sent to the Raspberry Pi, encoded and made available online. They are then collected and played successfully to the baby. In the future, we should explore sound treatment, so that the sound resembles more the sound from the womb. The prototype can also be simplified and the Raspberry Pi that accompanies the mother can be disposed of. Another option is to connect the Stemoscope directly to the Raspberry Pi and dispose of one of the Smartphones.

The side of the prototype in which the mother sees and listens to the baby was also implemented successfully. The material shows correct behavior and, by using the Raspicam, there are no compatibility issues with the Raspberry Pi. UV4L's web page provides a way for the mother to see the baby whenever she wants.

Implementation of the full prototype was successful. Both streams are simultaneously functional inside the Raspberry Pi by the incubator. Collecting and streaming the baby's video and audio does not interfere with receiving the womb sounds from IceCast and playing them to the baby using VLC.

The implementation of the app was only partially successful. The app manages to establish initial contact successfully through the signaling server. The negotiation stage also happens successfully and P2P communication is established. However, the stream only supports audio coming from the incubator, due to the incompatibilities between WebRTC implementations described in chapter 4. In the future, concluding the app should be a priority, so that the prototype can be fully tested.

Due to the pandemic situation it was not possible to test the prototype in a incubator. After completing the app and successfully establishing the full stream, testing the prototype with an incubator should be the next step.

We also do not know how the speakers will behave when connected to an incubator surface. They might not be suitable for application on an incubator.

The contact microphone collects a lot of noise, due to the fact that every sound wave coming in contact with its surface is collected. Therefore, collecting samples for proper sound treatment is important.

Treating the sound collected from the mother is also important. It should be as close as possible to what the baby would hear inside the womb and the Stemoscope does not provide that feature.

Bibliography

[1] Sociedade Portuguesa de Pediatria. *Dia Mundial da Prematuridade*. Visited on January 11th, 2021. URL: https://www.spp.pt/noticias/default.asp?IDN=372&op=2&ID=132.

[2] Kimberly Howard et al. 'Early Mother-Child Separation, Parenting, and Child Well-Being in EarlyHead Start Families, doi:10.1080/14616734.2010.488119'. In: (2011).

[3] P. Bremmer, J. Byers and E. Kiehl. In: *Journal of Obstetric, Gynecologic, and NeonatalNursing* (2003), pp. 447–454.

[4] Parga J.J. et al. *A description of externally recorded womb sounds in human subjects during gestation.* PLoS ONE 13(5): e0197045. URL: https://doi.org/10.1371/journal.%20pone.0197045.

[5] Devroey and Edmond M. 'Uterine Sound and Motion Simulation Device, doi:10.1080/14616734. 2010.488119'. In: *International Patent 20,130,096,368, issued* (April 18, 2013.).

[6] Keisuke Wakabayashi (Saitama) et al. 'Incubator, doi:10.1080/14616734.2010.488119'. In: *US Patent 10,245,199, issued* (Apr 2, 2019).

[7] Keisuke Wakabayashi (Saitama) et al. 'Baby Monitoring System'. In: *US Patent US8,094,013B1, issued* (Jan. 10, 2012).

[8] Ronald S. Kolarovic et al. 'Infant Incubator with Non-Contact Sensing and Monitoring'. In: *US Patent 6,679,830 B2, issued* (Feb. 6, 2002).

[9] *UV4L - Overview.* URL: https://www.linux-projects.org/uv4l/.

[10] *PulseAudio*. Visited on October 15th, 2021. URL: https://www.freedesktop.org/wiki/Software/PulseAudio/.

[11] Yev Kanivets. *Cross-platform mobile apps development in 2021: Xamarin vs React Native vs Flutter vs Kotlin Multiplatform*. Visited on October 14th, 2021. URL: https://medium.com/xorum-io/cross-platform-mobile-apps-development-in-2021-xamarin-vs-react-native-vs-flutter-vs-kotlin-ca8ea1f5a3e0.

[12] Visited on October 10th, 2021. URL: https://kotlinlang.org/.

[13] JetBrains. *Connect to platform-specific APIs*. Visited on October 17th, 2021. URL: https://kotlinlang.org/docs/kmm-connect-to-platform-specific-apis.html.

[14] Google Developers. *Kotlin coroutines*. Visited on October 17th, 2021. URL: https://developer.android.com/kotlin/coroutines.

[15] Alexandra Altvater. *Gradle vs. Maven: Performance, Compatibility, Builds, & More*. Visited on October 18th, 2021. URL: https://stackify.com/gradle-vs-maven/.

[16] Google Developers. *Real-time communication for the web*. Visited on October 25th, 2021. URL: https://webrtc.org/.

[17] Internet Engineering Task Force (IETF). ‘Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal’. In: *Standards Track* ISSN: 2070-1721 (2018). DOI: https://www.rfc-editor.org/rfc/rfc8445.

[18] MDN Web Docs. *WebRTC API*. Visited on October 14th, 2021. URL: https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API.

[19] Internet Engineering Task Force (IETF). ‘TCP Candidates with Interactive Connectivity Establishment (ICE)’. In: *Standards Track* ISSN: 2070-1721 (2012). DOI: https://datatracker.ietf.org/doc/html/rfc6544.

[20] Internet Engineering Task Force (IETF). ‘Trickle ICE: Incremental Provisioning of Candidates for the Interactive Connectivity Establishment (ICE) Protocol’. In: *Standards Track* ISSN: 2070-1721 (2021). DOI: https://datatracker.ietf.org/doc/html/rfc8838.

[21] Ian Hickson et al. ‘WebRTC 1.0: Real-time Communication Between Browsers’. In: (2011). DOI: https://w3c.github.io/webrtc-pc/archives/20151006/webrtc.html#state-definitions.

[22] Network Working Group. ‘SDP: Session Description Protocol’. In: *Standards Track* ISSN: 2070-1721 (2006). DOI: https://datatracker.ietf.org/doc/html/rfc4566.